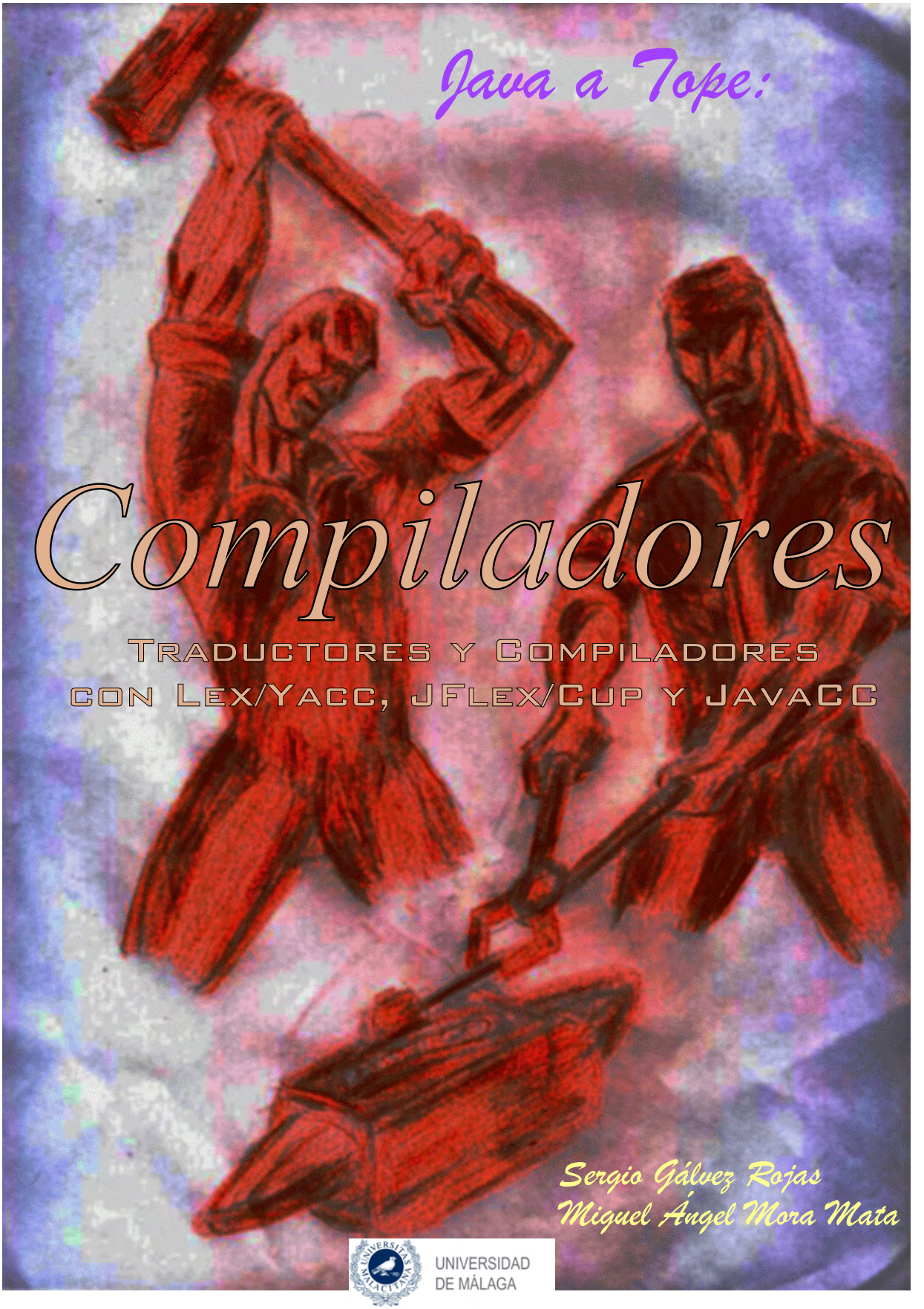

Java a Tope:
Compiladores
Traductores y Compiladores
con Lex/Yacc, JFlex/Cup y JavaCC
Sergio Gálvez Rojas
Miguel Ángel Mora Mata
UNIVERSIDAD
DE MÁLAGA

JAVA A TOPE:
TRADUCTORES Y COMPILADORES CON LEX/YACC, JFLEX/CUP Y JAVACC.
EDICIÓN ELECTRÓNICA

AUTORES: SERGIO GÁLVEZ ROJAS
MIGUEL ÁNGEL MORA MATA

ILUSTRACIÓN
DE PORTADA: JOSÉ MIGUEL GÁLVEZ ROJAS

Sun, el logotipo de Sun, Sun Microsystems y Java son marcas o marcas registradas de Sun Microsystems Inc. en los EE.UU. y otros países. El personaje de «Duke» es una marca de Sun Microsystems Inc.
PCLex y PCYacc son productos de Abraxas Software Inc.
JFlex está liberado con licencia GPL.
Cup está protegido por las licencias de código abierto, siendo compatible con la licencia GPL.
JavaCC está sujeto a la licencia BSD (Berkeley Software Distribution) de código abierto.

© 2005 por Sergio Gálvez Rojas

DEPÓSITO LEGAL: MA-185-2005
ISBN: 84-689-1037-6

Java a tope:

Compiladores

Traductores y Compiladores con Lex/Yacc, JFlex/cup y JavaCC

Sergio Gálvez Rojas
Doctor Ingeniero en Informática

Miguel Ángel Mora Mata
Ingeniero en Informática

Dpto. de Lenguajes y Ciencias de la Computación
E.T.S. de Ingeniería Informática
Universidad de Málaga

Índice

Prólogo vii

Capítulo 1 **Introducción** 1

1.1 Visión general 1
1.2 Concepto de traductor. 2
1.2.1 Tipos de traductores 2
1.2.1.1 Traductores del idioma 3
1.2.1.2 Compiladores 3
1.2.1.3 Intérpretes 3
1.2.1.4 Preprocesadores 4
1.2.1.5 Intérpretes de comandos 4
1.2.1.6 Ensambladores y macroensambladores 5
1.2.1.7 Conversores fuente-fuente 5
1.2.1.8 Compilador cruzado 6
1.2.2 Conceptos básicos relacionados con la traducción 6
1.2.2.1 Compilación, enlace y carga. 6
1.2.2.2 Pasadas de compilación 8
1.2.2.3 Compilación incremental 9
1.2.2.4 Autocompilador 10
1.2.2.5 Metacompilador 10
1.2.2.6 Descompilador 11
1.3 Estructura de un traductor 11
1.3.1 Construcción sistemática de compiladores 12
1.3.2 La tabla de símbolos 14
1.4 Ejemplo de compilación 15
1.4.1 Preprocesamiento 16
1.4.2 Etapa de análisis 16
1.4.2.1 Fase de análisis lexicográfico 17
1.4.2.2 Fase de análisis sintáctico 17
1.4.2.2.1 Compilación dirigida por sintaxis 18
1.4.2.3 Fase de análisis semántico 19
1.4.3 Etapa de síntesis 19
1.4.3.1 Fase de generación de código intermedio 19
1.4.3.2 Fase de optimización de código 20
1.4.3.3 Fase de generación de código máquina 20

Capítulo 2 **Análisis lexicográfico 23**
2.1 Visión general 23
2.2 Concepto de analizador léxico 23
2.2.1 Funciones del analizador léxico 24
2.2.2 Necesidad del analizador léxico 25
2.2.2.1 Simplificación del diseño 25
2.2.2.2 Eficiencia 27
2.2.2.3 Portabilidad 27
2.2.2.4 Patrones complejos 28
2.3 *Token*, patrón y lexema 28
2.3.1 Aproximaciones para construir un analizador lexicográfico . . . 30
2.4 El generador de analizadores lexicográficos: PCLex 31
2.4.1 Visión general 31
2.4.2 Creación de un analizador léxico 32
2.4.3 El lenguaje Lex 33
2.4.3.1 Premisas de Lex para reconocer lexemas 34
2.4.3.2 Caracteres especiales de Lex 34
2.4.3.3 Caracteres de sensibilidad al contexto 35
2.4.3.4 Estado léxicos 36
2.4.3.5 Área de definiciones y área de funciones 37
2.4.3.6 Funciones y variables suministradas por PCLex 38
2.5 El generador de analizadores lexicográficos JFlex 42
2.5.1 Ejemplo preliminar 42
2.5.2 Área de opciones y declaraciones 44
2.5.2.1 Opciones 44
2.5.2.1.1 Opciones de clase 44
2.5.2.1.2 Opciones de la función de análisis 44
2.5.2.1.3 Opciones de fin de fichero 45
2.5.2.1.4 Opciones de juego de caracteres 45
2.5.2.1.5 Opciones de contadores 46
2.5.2.2 Declaraciones. 46
2.5.2.2.1 Declaraciones de estados léxicos. 46
2.5.2.2.2 Declaraciones de reglas. 46
2.5.3 Área de reglas. 46
2.5.4 Funciones y variables de la clase Yylex 47

Capítulo 3 **Análisis sintáctico 49**
3.1 Visión general 49
3.2 Concepto de analizador sintáctico 49
3.3 Manejo de errores sintácticos 50
3.3.1 Ignorar el problema 51
3.3.2 Recuperación a nivel de frase 52
3.3.3 Reglas de producción adicionales 52

3.3.4 Corrección Global 52
3.4 Gramática utilizada por un analizador sintáctico 52
3.4.1 Derivaciones 53
3.4.2 Árbol sintáctico de una sentencia de un lenguaje 54
3.5 Tipos de análisis sintáctico 56
3.5.1 Análisis descendente con retroceso 57
3.5.2 Análisis descendente con funciones recursivas 60
3.5.2.1 Diagramas de sintaxis 61
3.5.2.2 Potencia de los diagramas de sintaxis 61
3.5.2.3 Correspondencia con flujos de ejecución 62
3.5.2.4 Ejemplo completo 63
3.5.2.5 Conclusiones sobre el análisis descendente con funciones recursivas 66
3.5.3 Análisis descendente de gramáticas LL(1) 67
3.5.4 Generalidades del análisis ascendente 70
3.5.4.1 Operaciones en un analizador ascendente 71
3.5.5 Análisis ascendente con retroceso 72
3.5.6 Análisis ascendente de gramáticas LR(1) 74
3.5.6.1 Consideraciones sobre el análisis LR(1) 79
3.5.6.1.1 Recursión a derecha o a izquierda 79
3.5.6.1.2 Conflictos 80
3.5.6.2 Conclusiones sobre el análisis LR(1) 81

Capítulo 4 **Gramáticas atribuidas 83**
4.1 Análisis semántico 83
4.1.1 Atributos y acciones semánticas 83
4.1.2 Ejecución de una acción semántica 86
4.2 Traducción dirigida por sintaxis 88
4.2.1 Definición dirigida por sintaxis 89
4.2.2 Esquema formal de una definición dirigida por sintaxis 89
4.2.2.1 Atributos sintetizados 91
4.2.2.2 Atributos heredados 92
4.2.2.3 Grafo de dependencias 93
4.2.2.4 Orden de evaluación 95
4.2.2.5 Gramática L-atribuida 96
4.2.2.6 Gramática S-atribuida 97
4.2.3 Esquemas de traducción 98
4.2.4 Análisis LALR con atributos 99
4.2.4.1 Conflictos reducir/reducir 100
4.2.4.2 Conflictos desplazar/reducir 100
4.3 El generador de analizadores sintácticos PCYacc 102
4.3.1 Formato de un programa Yacc 102
4.3.1.1 Área de definiciones 102

4.3.1.2 Creación del analizador léxico 103
4.3.1.3 Área de reglas 105
4.3.1.4 Área de funciones 106
4.3.2 Gestión de atributos 106
4.3.2.1 Acciones intermedias 110
4.3.3 Ambigüedad 110
4.3.4 Tratamiento de errores 113
4.3.5 Ejemplo final 116
4.4 El generador de analizadores sintácticos Cup 117
4.4.1 Diferencias principales 117
4.4.2 Sintaxis completa. 120
4.4.2.1 Especificación del paquete e importaciones. 120
4.4.2.2 Código de usuario. 120
4.4.2.3 Listas de símbolos 121
4.4.2.4 Asociatividad y precedencia. 121
4.4.2.5 Gramática. 122
4.4.3 Ejecución de Cup. 123
4.4.4 Comunicación con JFlex 124

Capítulo 5 **JavaCC 127**
5.1 Introducción 127
5.1.1 Características generales 127
5.1.2 Ejemplo preliminar. 128
5.2 Estructura de un programa en JavaCC 130
5.2.1 Opciones. 132
5.2.2 Área de *tokens* 134
5.2.2.1 Caracteres especiales para patrones JavaCC 135
5.2.2.2 Elementos accesibles en una acción léxica 137
5.2.2.3 La clase Token y el *token manager* 139
5.2.3 Área de funciones BNF 140
5.3 Gestión de atributos. 144
5.4 Gestión de errores. 146
5.4.1 Versiones antiguas de JavaCC 146
5.4.2 Versiones modernas de JavaCC 147
5.5 Ejemplo final 148
5.6 La utilidad JJTree 150
5.6.1 Características 150
5.6.2 Ejemplo Práctico 150
5.6.2.1 Renombrado de nodos mediante # 154
5.6.2.2 Construcción del Árbol 155
5.6.2.3 Tipos de Nodos 156
5.6.3 Ámbitos de Nodos y Acciones de Usuario 158
5.6.4 Manejo de Excepciones 160

5.6.5 *Node Scope Hooks* 160
5.6.6 El ciclo de vida de un nodo 163
5.6.7 *Visitor Support* 164
5.6.7.1 Parámetros de Ejecución 164
5.6.8 Estados en JTree 165
5.6.8.1 Objetos Node 165
5.6.8.2 Pasos de Ejecución 166
5.6.9 Manual de Referencia de JJDoc 167

Capítulo 6 **Tabla de símbolos 171**
6.1 Visión general 171
6.2 Información sobre los identificadores de usuario 171
6.3 Consideraciones sobre la tabla de símbolos 173
6.4 Ejemplo: una calculadora con variables 175
6.4.1 Interfaz de la tabla de símbolos 176
6.4.2 Solución con Lex/Yacc 176
6.4.3 Solución con JFlex/Cup 179
6.4.4 Solución con JavaCC 182

Capítulo 7 **Gestión de tipos 187**
7.1 Visión general 187
7.2 Compatibilidad nominal, estructural y funcional 188
7.3 Gestión de tipos primitivos 190
7.3.1 Gramática de partida 190
7.3.2 Pasos de construcción 192
7.3.2.1 Propuesta de un ejemplo 192
7.3.2.2 Definición de la tabla de símbolos 193
7.3.2.3 Asignación de atributos 193
7.3.2.3.1 Atributos de terminales 194
7.3.2.3.2 Atributos de no terminales 195
7.3.2.4 Acciones semánticas 196
7.3.3 Solución con Lex/Yacc 198
7.3.4 Solución con JFlex/Cup 203
7.3.5 Solución con JavaCC 207
7.4 Gestión de tipos complejos 210
7.4.1 Objetivos y propuesta de un ejemplo 211
7.4.2 Pasos de construcción 212
7.4.2.1 Gramática de partida 212
7.4.2.2 Gestión de atributos 214
7.4.2.3 Implementación de una pila de tipos 215
7.4.2.4 Acciones semánticas 217
7.4.3 Solución con Lex/Yacc 220
7.4.4 Solución con JFlex/Cup 224

7.4.5 Solución con JavaCC 229

Capítulo 8 **Generación de código 235**
8.1 Visión general 235
8.2 Código de tercetos 237
8.3 Una calculadora simple compilada 238
8.3.1 Pasos de construcción 239
8.3.1.1 Propuesta de un ejemplo 239
8.3.1.2 Gramática de partida 239
8.3.1.3 Consideraciones importantes sobre el traductor 240
8.3.1.4 Atributos necesarios 242
8.3.1.5 Acciones semánticas 244
8.3.2 Solución con Lex/Yacc 244
8.3.3 Solución con JFlex/Cup 246
8.3.4 Solución con JavaCC 247
8.4 Generación de código en sentencias de control 250
8.4.1 Gramática de partida 250
8.4.2 Ejemplos preliminares 254
8.4.3 Gestión de condiciones 257
8.4.3.1 Evaluación de condiciones usando cortocircuito 259
8.4.3.1.1 Condiciones simples 259
8.4.3.1.2 Condiciones compuestas 261
8.4.4 Gestión de sentencias de control de flujo 265
8.4.4.1 Sentencia IF-THEN-ELSE 266
8.4.4.2 Sentencia WHILE 268
8.4.4.3 Sentencia REPEAT 269
8.4.4.4 Sentencia CASE 272
8.4.5 Solución con Lex/Yacc 276
8.4.6 Solución con JFlex/Cup 281
8.4.7 Solución con JavaCC 286

Capítulo 9 **Gestión de memoria en tiempo de ejecución . . . 293**
9.1 Organización de la memoria durante la ejecución 293
9.2 Zona de código 294
9.2.1 *Overlays* 294
9.3 Zona de datos 295
9.3.1 Zona de Datos de Tamaño Fijo 296
9.3.2 Pila (*Stack*) 298
9.3.3 Montón (*heap*) 303

Prólogo

La construcción de un compilador es una de las tareas más gratas con las que un informático puede encontrarse a lo largo de su carrera profesional. Aunque no resulta sensato pensar que una labor tal pueda formar parte de la actividad cotidiana de la mayoría de estos profesionales, sí es cierto que, con cierta frecuencia, suele aparecer la necesidad de analizar un fichero de texto que contiene información distribuida según algún patrón reconocible: ficheros XML, ficheros de inicialización .ini, ficheros con comandos del sistema operativo, los propios programas fuente, etc.

Es más, el concepto de análisis de un texto puede ser útil para ahorrar trabajo en situaciones que están fuera del ámbito profesional informático, como por ejemplo la generación de índices analíticos en archivos de texto escritos con procesadores que no admiten esta opción.

Pero a pesar de la indudable utilidad de los conceptos generales relacionados con la teoría y práctica de los análisis de textos, el crear un compilador introduce al informático en un nuevo mundo en el que el control sobre el ordenador es absoluto y le lleva al paroxismo de la omnipotencia sobre la máquina. Y es que resulta imprescindible un pleno conocimiento de todos los conceptos intrínsecos al ordenador: dispositivos de E/S, código máquina utilizado por el microprocesador, comunicación a bajo nivel entre éste y el resto de sus periféricos, distribución de la memoria, mecanismos de uso de memoria virtual, etc.

No obstante, el lector no debe asustarse ante el poder que este libro le va a conceder, ni tampoco ante los grandes conocimientos que exige el control de ese poder. Un compilador es una programa que da mucho juego en el sentido de que puede apoyarse sobre otras herramientas ya existentes, lo que facilita enormemente el aprendizaje de su construcción y permite su estudio de forma gradual y pausada.

En el volumen que el lector tiene en sus manos se aúna una gran cantidad de información orientada a la construcción de compiladores, pero en la que se presta especial atención a las herramientas destinadas a facilitar la labor de reconocimiento de textos que siguen una determinada gramática y léxico. Es por ello que los ejemplos propuestos se resuelven desde una perspectiva ambivalente: de un lado mediante Lex y Yacc, ya que por motivos históricos constituyen el pilar principal de apoyo al reconocimiento léxico y sintáctico; y de otro lado mediante JavaCC, lo que nos introduce tanto en la utilización del lenguaje Java como medio para construir compiladores, como en los mecanismos basados en notación BNF para dicha construcción. Con el objetivo de centrar nuestra atención en el lenguaje Java, los ejemplos también son resueltos mediante las herramientas FLEX y Cup que constituyen versiones adaptadas de Lex y Yacc para generar código Java orientado a objetos.

Hemos escogido JavaCC frente a otras opciones como SableCC, CoCo/Java, etc. (véase la página http://catalog.compilertools.net/java.html) por dos motivos principales. Primero, JavaCC ha sido apoyado por Sun Microsystems hasta hace bien poco (hoy por hoy Sun Microsystems no apoya a ninguna de estas utilidades, ya que se hallan lo suficientemente maduras como para proseguir solas su camino). Y segundo, se trata de una de las herramientas más potentes basadas en una notación sustancialmente diferente a la empleada por Lex y Yacc, lo cual enriquecerá nuestra percepción de este excitante mundo: la construcción de compiladores.

Capítulo 1

Introducción

1.1 Visión general

Uno de los principales mecanismos de comunicación entre un ordenador y una persona viene dado por el envío y recepción de mensajes de tipo textual: el usuario escribe una orden mediante el teclado, y el ordenador la ejecuta devolviendo como resultado un mensaje informativo sobre las acciones llevadas a cabo.

Aunque la evolución de los ordenadores se encuentra dirigida actualmente hacia el empleo de novedosas y ergonómicas interfaces de usuario (como el ratón, las pantallas táctiles, las tabletas gráficas, etc.), podemos decir que casi todas las acciones que el usuario realiza sobre estas interfaces se traducen antes o después a secuencias de comandos que son ejecutadas como si hubieran sido introducidas por teclado. Por otro lado, y desde el punto de vista del profesional de la Informática, el trabajo que éste realiza sobre el ordenador se encuentra plagado de situaciones en las que se produce una comunicación textual directa con la máquina: utilización de un intérprete de comandos (*shell*), construcción de ficheros de trabajo por lotes, programación mediante diversos lenguajes, etc. Incluso los procesadores de texto como WordPerfect y MS Word almacenan los documentos escritos por el usuario mediante una codificación textual estructurada que, cada vez que se abre el documento, es reconocida, recorrida y presentada en pantalla.

Por todo esto, ningún informático que se precie puede esquivar la indudable necesidad de conocer los entresijos de la herramienta que utiliza durante su trabajo diario y sobre la que descansa la interacción hombre-máquina: el **traductor**.

Existe una gran cantidad de situaciones en las que puede ser muy útil conocer cómo funcionan las distintas partes de un compilador, especialmente aquélla que se encarga de trocear los textos fuentes y convertirlos en frases sintácticamente válidas. Por ejemplo, una situación de aparente complejidad puede presentársenos si se posee un documento de MS Word que procede de una fusión con una base de datos y se quiere, a partir de él, obtener la B.D. original. ¿Cómo solucionar el problema? Pues basándose en que la estructura del documento está formada por bloques que se repiten; la solución podría ser:

- Convertir el documento a formato texto puro.
- Procesar dicho texto con un traductor para eliminar los caracteres superfluos y dar como resultado otro texto en el que cada campo de la tabla de la B.D. está entre comillas.
- El texto anterior se importa con cualquier SGBD.

Otras aplicaciones de la construcción de traductores pueden ser la creación de preprocesadores para lenguajes que no lo tienen (por ejemplo, para trabajar fácilmente con SQL en C, se puede hacer un preprocesador para introducir SQL inmerso), o incluso la conversión del carácter ASCII 10 (LF) en "<br>" de HTML para pasar texto a la web.

En este capítulo, se introduce la construcción de un compilador y se describen sus componentes, el entorno en el que estos trabajan y algunas herramientas de software que facilitan su construcción.

1.2 Concepto de traductor.

Un traductor se define como **un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino produciendo, si cabe, mensajes de error**. Los traductores engloban tanto a los compiladores (en los que el lenguaje destino suele ser código máquina) como a los intérpretes (en los que el lenguaje destino está constituido por las acciones atómicas que puede ejecutar el intérprete). La figura 1.1 muestra el esquema básico que compone a un compilador/intérprete.

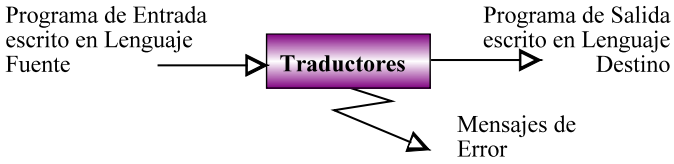


Figura 1.1. Esquema preliminar de un traductor

Es importante destacar la velocidad con la que hoy en día se puede construir un compilador. En la década de 1950, se consideró a los traductores como programas notablemente difíciles de escribir. El primer compilador de Fortran (*Formula Translator*), por ejemplo, necesitó para su implementación el equivalente a 18 años de trabajo individual (realmente no se tardó tanto puesto que el trabajo se desarrolló en equipo). Hasta que la teoría de autómatas y lenguajes formales no se aplicó a la creación de traductores, su desarrollo ha estado plagado de problemas y errores. Sin embargo, hoy día un compilador básico puede ser el proyecto fin de carrera de cualquier estudiante universitario de Informática.

1.2.1 Tipos de traductores

Desde los orígenes de la computación, ha existido un abismo entre la forma en que las personas expresan sus necesidades y la forma en que un ordenador es capaz de interpretar instrucciones. Los traductores han intentado salvar este abismo para facilitarle el trabajo a los humanos, lo que ha llevado a aplicar la teoría de autómatas a diferentes campos y áreas concretas de la informática, dando lugar a los distintos tipos de traductores que veremos a continuación.

1.2.1.1 Traductores del idioma

Traducen de un idioma dado a otro, como por ejemplo del inglés al español. Este tipo de traductores posee multitud de problemas, a saber:

- Necesidad de inteligencia artificial y problema de las frases hechas. El problema de la inteligencia artificial es que tiene mucho de artificial y poco de inteligencia, por lo que en la actualidad resulta casi imposible traducir frases con un sentido profundo. Como anécdota, durante la guerra fría, en un intento por realizar traducciones automáticas del ruso al inglés y viceversa, se puso a prueba un prototipo introduciendo el texto en inglés: "El espíritu es fuerte pero la carne es débil" cuya traducción al ruso se pasó de nuevo al inglés para ver si coincidía con el original. Cual fue la sorpresa de los desarrolladores cuando lo que se obtuvo fue: "El vino está bueno pero la carne está podrida" (en inglés *spirit* significa tanto espíritu como alcohol). Otros ejemplos difíciles de traducir lo constituyen las frases hechas como: "Piel de gallina", "por si las moscas", "molar mazo", etc.
- Difícil formalización en la especificación del significado de las palabras.
- Cambio del sentido de las palabras según el contexto. Ej: "por decir aquello, se llevó una galleta".
- En general, los resultados más satisfactorios en la traducción del lenguaje natural se han producido sobre subconjuntos restringidos del lenguaje. Y aún más, sobre subconjuntos en los que hay muy poco margen de ambigüedad en la interpretación de los textos: discursos jurídicos, documentación técnica, etc.

1.2.1.2 Compiladores

Es aquel traductor que tiene como entrada una sentencia en lenguaje formal y como salida tiene un fichero ejecutable, es decir, realiza una traducción de un código de alto nivel a código máquina (también se entiende por compilador aquel programa que proporciona un fichero objeto en lugar del ejecutable final).

1.2.1.3 Intérpretes

Es como un compilador, solo que la salida es una ejecución. El programa de entrada se reconoce y ejecuta a la vez. No se produce un resultado físico (código máquina) sino lógico (una ejecución). Hay lenguajes que sólo pueden ser interpretados, como p.ej. SNOBOL (*StriNg Oriented SimBOlyc Language*), LISP (*LISt Processing*), algunas versiones de BASIC (*Beginner's All-purpose Symbolic Instruction Code*), etc.

Su principal ventaja es que permiten una fácil depuración. Entre los inconvenientes podemos citar, en primer lugar, la lentitud de ejecución , ya que al ejecutar a la vez que se traduce no puede aplicarse un alto grado de optimización; por ejemplo, si el programa entra en un bucle y la optimización no está muy afinada, las mismas instrucciones se interpretarán y ejecutarán una y otra vez, enlenteciendo la ejecución del programa. Otro inconveniente es que durante la ejecución, el intérprete debe residir en memoria, por lo que consumen más recursos.

Además de que la traducción optimiza el programa acercándolo a la máquina, los lenguajes interpretados tienen la característica de que permiten construir programas que se pueden modificar a sí mismos.

Algunos lenguajes intentan aunar las ventajas de los compiladores y de los intérpretes y evitar sus desventajas; son los lenguajes pseudointerpretados. En estos, el programa fuente pasa por un pseudocompilador que genera un pseudoejecutable. Para ejecutar este pseudoejecutable se le hace pasar por un motor de ejecución que lo interpreta de manera relativamente eficiente. Esto tiene la ventaja de la portabilidad, ya que el pseudoejecutable es independiente de la máquina en que vaya a ejecutarse, y basta con que en dicha máquina se disponga del motor de ejecución apropiado para poder interpretar cualquier pseudoejecutable. El ejemplo actual más conocido lo constituye el lenguaje Java; también son pseudointerpretadas algunas versiones de Pascal y de COBOL (*COmmon Bussiness Oriented Language*). La figura 1.2 muestra los pasos a seguir en estos lenguajes para obtener una ejecución.

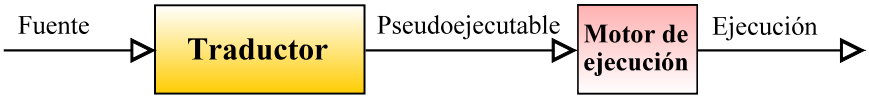


Figura 1.2. Esquema de traducción/ejecución de un programa interpretado

1.2.1.4 Preprocesadores

Permiten modificar el programa fuente antes de la verdadera compilación. Hacen uso de macroinstrucciones y directivas de compilación. Por ejemplo, en lenguaje C, el preprocesador sustituye la directiva `#include Uno.c` por el código completo que contiene el fichero "Uno.c", de manera que cuando el compilador comienza su ejecución se encuentra con el código ya insertado en el programa fuente (la figura 1.3 ilustra esta situación). Algunas otras directivas de preprocesamiento permiten compilar trozos de códigos opcionales (lenguajes C y Clipper): `#fi`, `#ifdef`, `#define`, `#ifndef`, etc. Los preprocesadores suelen actuar de manera transparente para el programador, pudiendo incluso considerarse que son una fase preliminar del compilador.

1.2.1.5 Intérpretes de comandos

Un intérprete de comandos traduce sentencias simples a invocaciones a programas de una biblioteca. Se utilizan especialmente en los sistemas operativos (la *shell* de Unix es un intérprete de comandos). Los programas invocados pueden residir en el *kernel* (núcleo) del sistema o estar almacenados en algún dispositivo externo como rutinas ejecutables que se traen a memoria bajo demanda.

Por ejemplo, si bajo MS-DOS se teclea el comando `copy` se ejecutará la función de copia de ficheros del sistema operativo, que se encuentra residente en memoria.

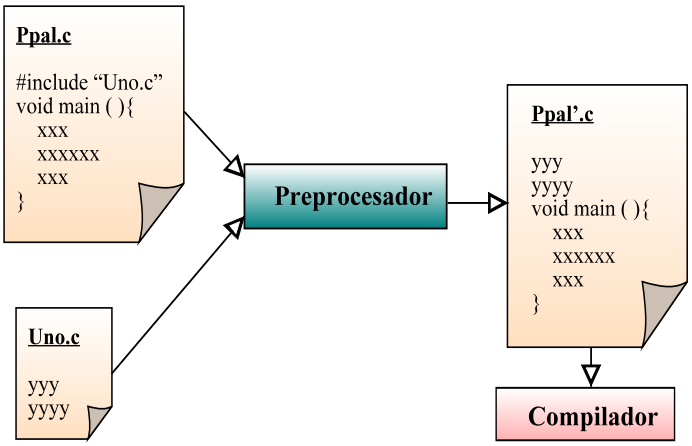


Figura 1.3. Funcionamiento de la directiva de preprocesamiento #include en lenguaje C

1.2.1.6 Ensambladores y macroensambladores

Son los pioneros de los compiladores, ya que en los albores de la informática, los programas se escribían directamente en código máquina, y el primer paso hacia los lenguajes de alto nivel lo constituyen los ensambladores. En lenguaje ensamblador se establece una relación biunívoca entre cada instrucción y una palabra mnemotécnica, de manera que el usuario escribe los programas haciendo uso de los mnemotécnicos, y el ensamblador se encarga de traducirlo a código máquina puro. De esta manera, los ensambladores suelen producir directamente código ejecutable en lugar de producir ficheros objeto.

Un ensamblador es un compilador sencillo, en el que el lenguaje fuente tiene una estructura tan sencilla que permite la traducción de cada sentencia fuente a una única instrucción en código máquina. Al lenguaje que admite este compilador también se le llama lenguaje ensamblador. En definitiva, existe una correspondencia uno a uno entre las instrucciones ensamblador y las instrucciones máquina. Ej:

Instrucción ensamblador: LD HL, #0100
Código máquina generado: 65h.00h.01h

Por otro lado, existen ensambladores avanzados que permiten definir macroinstrucciones que se pueden traducir a varias instrucciones máquina. A estos programas se les llama macroensambladores, y suponen el siguiente paso hacia los lenguajes de alto nivel. Desde un punto de vista formal, un macroensamblador puede entenderse como un ensamblador con un preprocesador previo.

1.2.1.7 Conversores fuente-fuente

Permiten traducir desde un lenguaje de alto nivel a otro lenguaje de alto nivel,

con lo que se consigue una mayor portabilidad en los programas de alto nivel.

Por ejemplo, si un ordenador sólo dispone de un compilador de Pascal, y queremos ejecutar un programa escrito para otra máquina en COBOL, pues un conversor de COBOL a Pascal solucionará el problema. No obstante el programa fuente resultado puede requerir retoques manuales debido a diversos motivos:

- En situaciones en que el lenguaje destino carece de importantes características que el lenguaje origen sí tiene. Por ejemplo un conversor de Java a C, necesitaría modificaciones ya que C no tiene recolector de basura.
- En situaciones en que la traducción no es inteligente y los programas destino son altamente ineficientes.

1.2.1.8 Compilador cruzado

Es un compilador que genera código para ser ejecutado en otra máquina. Se utilizan en la fase de desarrollo de nuevos ordenadores. De esta manera es posible, p.ej., construir el sistema operativo de un nuevo ordenador recurriendo a un lenguaje de alto nivel, e incluso antes de que dicho nuevo ordenador disponga siquiera de un compilador.

Nótese también que, para facilitar el desarrollo de software de un nuevo ordenador, uno de los primeros programas que se deben desarrollar para éste es, precisamente, un compilador de algún lenguaje de alto nivel.

1.2.2 Conceptos básicos relacionados con la traducción

Vamos a estudiar a continuación diversa terminología relacionada con el proceso de compilación y de construcción de compiladores.

1.2.2.1 Compilación, enlace y carga.

Estas son las tres fases básicas que hay que seguir para que un ordenador ejecute la interpretación de un texto escrito mediante la utilización de un lenguaje de alto nivel. Aunque este libro se centrará exclusivamente en la primera fase, vamos a ver en este punto algunas cuestiones relativas al proceso completo.

Por regla general, el compilador no produce directamente un fichero ejecutable, sino que el código generado se estructura en módulos que se almacenan en un fichero objeto. Los ficheros objeto poseen información relativa tanto al código máquina como a una tabla de símbolos que almacena la estructura de las variables y tipos utilizados por el programa fuente. La figura 1.4 muestra el resultado real que produce un compilador.

Figura 1.4. Entrada y salida de un compilador real

Pero, ¿por qué no se genera directamente un fichero ejecutable? Sencillamente, para permitir la compilación separada, de manera que varios programadores puedan desarrollar simultáneamente las partes de un programa más grande y, lo que es más importante, puedan compilarlos independientemente y realizar una depuración en paralelo. Una vez que cada una de estas partes ha generado su correspondiente fichero objeto, estos deben fusionarse para generar un solo ejecutable.

Como se ha comentado, un fichero objeto posee una estructura de módulos también llamados registros. Estos registros tienen longitudes diferentes dependiendo de su tipo y cometido. Ciertos tipos de estos registros almacenan código máquina, otros poseen información sobre las variables globales, y otros incluyen información sobre los objetos externos (p. ej, variables que se supone que están declaradas en otro ficheros. El lenguaje C permite explícitamente esta situación mediante el modificador "extern").

Durante la fase de enlace, el enlazador o *linker* resuelve las referencias cruzadas, (así se llama a la utilización de objetos externos), que pueden estar declarados en otros ficheros objeto, o en librerías (ficheros con extensión "lib" o "dll"), engloba en un único bloque los distintos registros que almacenan código máquina, estructura el bloque de memoria destinado a almacenar las variables en tiempo de ejecución y genera el ejecutable final incorporando algunas rutinas adicionales procedentes de librerías, como por ejemplo las que implementan funciones matemáticas o de e/s básicas. La figura 1.5 ilustra este mecanismo de funcionamiento.

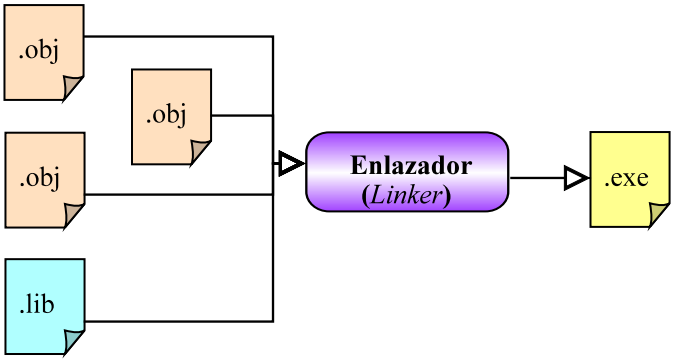


Figura 1.5. Funcionamiento de un enlazador

De esta manera, el bloque de código máquina contenido en el fichero ejecutable es un código reubicable, es decir, un código que en su momento se podrá ejecutar en diferentes posiciones de memoria, según la situación de la misma en el momento de la ejecución. Según el modelo de estructuración de la memoria del microprocesador, este código se estructura de diferentes formas. Lo más usual es que el fichero ejecutable esté dividido en segmentos: de código, de datos, de pila de datos, etc.

Cuando el enlazador construye el fichero ejecutable, asume que cada segmento va a ser colocado en la dirección 0 de la memoria. Como el programa va a estar dividido en segmentos, las direcciones a que hacen referencia las instrucciones dentro de cada segmento (instrucciones de cambio de control de flujo, de acceso a datos, etc.), no se tratan como absolutas, sino que son direcciones relativas a partir de la dirección base en que sea colocado cada segmento en el momento de la ejecución. El cargador carga el fichero .exe, coloca sus diferentes segmentos en memoria (donde el sistema operativo le diga que hay memoria libre para ello) y asigna los registros base a sus posiciones correctas, de manera que las direcciones relativas funcionen correctamente. La figura 1.6 ilustra el trabajo que realiza un cargador de programas.

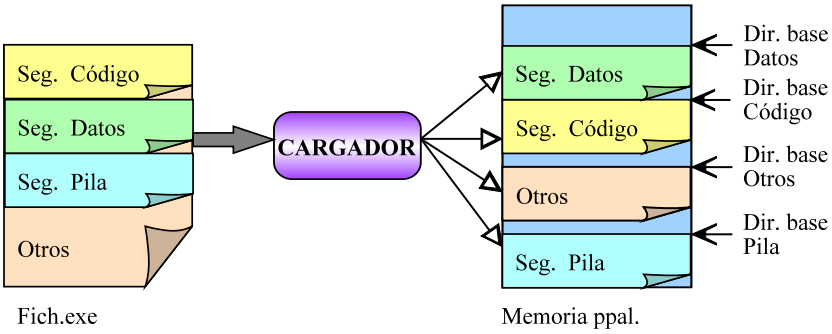


Figura 1.6. Labor realizada por el cargador. El cargador suele ser parte del sistema operativo

Cada vez que una instrucción máquina hace referencia a una dirección de memoria (partiendo de la dirección 0), el microprocesador se encarga automáticamente de sumar a dicha dirección la dirección absoluta de inicio de su segmento. Por ejemplo para acceder a la variable x almacenada en la dirección 1Fh, que se encuentra en el segmento de datos ubicado en la dirección 8A34h, la instrucción máquina hará referencia a 1Fh, pero el microprocesador la traducirá por 8A34h+1Fh dando lugar a un acceso a la dirección 8A53h: dir absoluta del segmento en memoria + dir relativa de x en el segmento = dir absoluta de x en memoria.

1.2.2.2 Pasadas de compilación

Es el número de veces que un compilador debe leer el programa fuente para generar el código. Hay algunas situaciones en las que, para realizar la compilación, no es suficiente con leer el fichero fuente una sola vez. Por ejemplo, en situaciones en las que existe recursión indirecta (una función A llama a otra B y la B llama a la A). Cuando se lee el cuerpo de A, no se sabe si B está declarada más adelante o se le ha olvidado al programador; o si lo está, si los parámetros reales coinciden en número y tipo con los formales o no. Es más, aunque todo estuviera correcto, aún no se ha generado el código para B, luego no es posible generar el código máquina

correspondiente a la invocación de B puesto que no se sabe su dirección de comienzo en el segmento de código. Por todo esto, en una pasada posterior hay que controlar los errores y rellenar los datos que faltan.

Diferentes compiladores y lenguajes solucionan este problema de maneras distintas. Una solución consiste en hacer dos o más pasadas de compilación, pero ello consume demasiado tiempo puesto que las operaciones de e/s son, hoy por hoy, uno de los puntos fundamentales en la falta de eficiencia de los programas (incluidos los compiladores). Otra solución pasa por hacer una sola pasada de compilación y modificar el lenguaje obligando a hacer las declaraciones de funciones recursivas indirectas antes de su definición. De esta manera, lenguajes como Pascal o Modula-2 utilizan la palabra reservada FORWARD para ello: FORWARD precede la declaración de B, a continuación se define A y por último se define B. Lenguajes como C también permiten el no tener que declarar una función si ésta carece de parámetros y devuelve un entero.

Otros lenguajes dan por implícito el FORWARD, y si aún no se han encontrado aquéllo a que se hace referencia, continúan, esperando que el *linker* resuelva el problema, o emita el mensaje de error.

Actualmente, cuando un lenguaje necesita hacer varias pasadas de compilación, suele colocar en memoria una representación abstracta del fichero fuente, de manera que las pasadas de compilación se realizan sobre dicha representación en lugar de sobre el fichero de entrada, lo que soluciona el problema de la ineficiencia debido a operaciones de e/s.

1.2.2.3 Compilación incremental

Cuando se desarrolla un programa fuente, éste se recompila varias veces hasta obtener una versión definitiva libre de errores. Pues bien, en una compilación incremental sólo se recompilan las modificaciones realizadas desde la última compilación. Lo ideal es que sólo se recompilen aquellas partes que contenían los errores o que, en general, hayan sido modificadas, y que el código generado se reinserte con cuidado en el fichero objeto generado en la última compilación. Sin embargo esto es muy difícil de conseguir y no suele ahorrar tiempo de compilación más que en casos muy concretos.

La compilación incremental se puede llevar a cabo con distintos grados de afinación. Por ejemplo, si se olvida un ';' en una sentencia, se podría generar un fichero objeto transitorio parcial. Si se corrige el error y se añade el ';' que falta y se recompila, un compilador incremental puede funcionar a varios niveles

- A nivel de carácter: se recompila el ';' y se inserta en el fichero objeto la sentencia que faltaba.
- A nivel de sentencia: si el ';' faltaba en la línea 100 sólo se compila la línea 100 y se actualiza el fichero objeto.
- A nivel de bloque: si el ';' faltaba en un procedimiento o bloque sólo se compila ese bloque y se actualiza el fichero objeto.

- A nivel de fichero fuente: si la aplicación completa consta de 15 ficheros fuente, y solo se modifica 1(al que le faltaba el ';'), sólo se compila aquél al que se le ha añadido el ';', se genera por completo su fichero objeto y luego se enlazan todos juntos para obtener el ejecutable.

Lo ideal es que se hiciese eficientemente a nivel de sentencia, pero lo normal es encontrarlo a nivel de fichero. La mayoría de los compiladores actuales realizan una compilación incremental a este nivel.

Cuando un compilador no admite compilación incremental, suelen suministrar una herramienta externa (como RMAKE en caso de Clipper, MAKE en algunas versiones de C) en la que el programador indica las dependencias entre ficheros, de manera que si se recompila uno, se recompilan todos los que dependen de aquél. Estas herramientas suelen estar diseñadas con un propósito más general y también permiten enlaces condicionales.

1.2.2.4 Autocompilador

Es un compilador escrito en el mismo lenguaje que compila (o parecido). Normalmente, cuando se extiende entre muchas máquinas diferentes el uso de un compilador, y éste se desea mejorar, el nuevo compilador se escribe utilizando el lenguaje del antiguo, de manera que pueda ser compilado por todas esas máquinas diferentes, y dé como resultado un compilador más potente de ese mismo lenguaje.

1.2.2.5 Metacompilador

Este es uno de los conceptos más importantes con los que vamos a trabajar. Un metacompilador es un compilador de compiladores. Se trata de un programa que acepta como entrada la descripción de un lenguaje y produce el compilador de dicho lenguaje. Hoy por hoy no existen metacompiladores completos, pero sí parciales en los que se acepta como entrada una gramática de un lenguaje y se genera un autómata que reconoce cualquier sentencia del lenguaje . A este autómata podemos añadirle código para completar el resto del compilador. Ejemplos de metacompiladores son: Lex, YACC, FLex, Bison, JavaCC, JLex, Cup, PCCTS, MEDISE, etc.

Los metacompiladores se suelen dividir entre los que pueden trabajar con gramáticas de contexto libre y los que trabajan con gramáticas regulares. Los primeros se dedican a reconocer la sintaxis del lenguaje y los segundos trocean los ficheros fuente y lo dividen en palabras.

PCLex es un metacompilador que genera la parte del compilador destinada a reconocer las palabras reservadas. PCYACC es otro metacompilador que genera la parte del compilador que informa sobre si una sentencia del lenguaje es válida o no. JavaCC es un metacompilador que aúna el reconocimiento de palabras reservadas y la aceptación o rechazo de sentencias de un lenguaje. PCLex y PCYACC generan código C y admiten descripciones de un lenguaje mediante gramáticas formales, mientras que JavaCC produce código Java y admite descripciones de lenguajes expresadas en notación BNF (*Backus-Naur Form*). Las diferencias entre ellas se irán estudiando en

temas posteriores.

1.2.2.6 Descompilador

Un descompilador realiza una labor de traducción inversa, esto es, pasa de un código máquina (programa de salida) al equivalente escrito en el lenguaje que lo generó (programa fuente). Cada descompilador trabaja con un lenguaje de alto nivel concreto.

La descompilación suele ser una labor casi imposible, porque al código máquina generado casi siempre se le aplica una optimización en una fase posterior, de manera que un mismo código máquina ha podido ser generado a partir de varios códigos fuente. Por esto, sólo existen descompiladores de aquellos lenguajes en los que existe una relación biyectiva entre el código destino y el código fuente, como sucede con los desensambladores, en los que a cada instrucción máquina le corresponde una y sólo una instrucción ensamblador.

Los descompiladores se utilizan especialmente cuando el código máquina ha sido generado con opciones de depuración, y contiene información adicional de ayuda al descubrimiento de errores (puntos de ruptura, seguimiento de trazas, opciones de visualización de variables, etc.).

También se emplean cuando el compilador no genera código máquina puro, sino pseudocódigo para ser ejecutado a través de un pseudointérprete. En estos casos suele existir una relación biyectiva entre las instrucciones del pseudocódigo y las construcciones sintácticas del lenguaje fuente, lo que permite reconstruir un programa de alto nivel a partir del de un bloque de pseudocódigo.

1.3 Estructura de un traductor

Un traductor divide su labor en dos etapas: una que analiza la entrada y genera estructuras intermedias y otra que sintetiza la salida a partir de dichas estructuras. Por tanto, el esquema de un traductor pasa de ser el de la figura 1.1, a ser el de la figura 1.7.

Básicamente los objetivos de la etapa de análisis son: a) controlar la corrección del programa fuente, y b) generar las estructuras necesarias para comenzar la etapa de síntesis.

Para llevar esto a cabo, la etapa de análisis consta de las siguientes fases:

- Análisis lexicográfico. Divide el programa fuente en los componentes básicos del lenguaje a compilar. Cada componente básico es una subsecuencia de

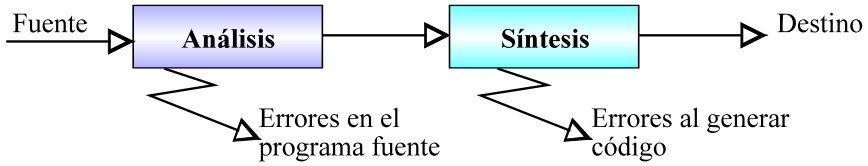


Figura 1.7. Esquema por etapas de un traductor

caracteres del programa fuente, y pertenece a una categoría gramatical: números, identificadores de usuario (variables, constantes, tipos, nombres de procedimientos, ...), palabras reservadas, signos de puntuación, etc.

- ✎ Análisis sintáctico. Comprueba que la estructura de los componentes básicos sea correcta según las reglas gramaticales del lenguaje que se compila.
- ✎ Análisis semántico. Comprueba que el programa fuente respeta las directrices del lenguaje que se compila (todo lo relacionado con el significado): chequeo de tipos, rangos de valores, existencia de variables, etc.

Cualquiera de estas tres fases puede emitir mensajes de error derivados de fallos cometidos por el programador en la redacción de los textos fuente. Mientras más errores controle un compilador, menos problemas dará un programa en tiempo de ejecución. Por ejemplo, el lenguaje C no controla los límites de un array, lo que provoca que en tiempo de ejecución puedan producirse comportamientos del programa de difícil explicación.

La etapa de síntesis construye el programa objeto deseado (equivalente semánticamente al fuente) a partir de las estructuras generadas por la etapa de análisis. Para ello se compone de tres fases fundamentales:

- ✎ Generación de código intermedio. Genera un código independiente de la máquina muy parecido al ensamblador. No se genera código máquina directamente porque así es más fácil hacer pseudocompiladores y además se facilita la optimización de código independientemente del microprocesador.
- ✎ Generación del código máquina. Crea un bloque de código máquina ejecutable, así como los bloques necesarios destinados a contener los datos.
- ✎ Fase de optimización. La optimización puede realizarse sobre el código intermedio (de forma independiente de las características concretas del microprocesador), sobre el código máquina, o sobre ambos. Y puede ser una aislada de las dos anteriores, o estar integrada con ellas.

1.3.1 Construcción sistemática de compiladores

Con frecuencia, las fases anteriores se agrupan en una **etapa inicial** (*front-end*) y una **etapa final** (*back- end*). La etapa inicial comprende aquellas fases, o partes de fases, que dependen exclusivamente del lenguaje fuente y que son independientes de la máquina para la cual se va a generar el código. En la etapa inicial se integran los análisis léxicos y sintácticos, el análisis semántico y la generación de código intermedio. La etapa inicial también puede hacer cierta optimización de código e incluye además, el manejo de errores correspondiente a cada una de esas fases.

La etapa final incluye aquellas fases del compilador que dependen de la máquina destino y que, en general, no dependen del lenguaje fuente sino sólo del lenguaje intermedio. En esta etapa, se encuentran aspectos de la fase de generación de

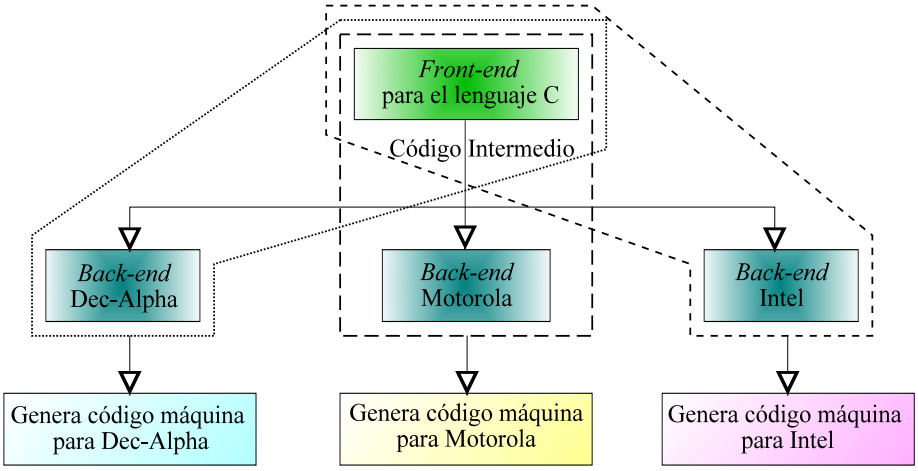


Figura 1.8. Construcción de tres compiladores de C reutilizando un *front-end*

código, además de su optimización, junto con el manejo de errores necesario y el acceso a las estructuras intermedias que haga falta.

Se ha convertido en una práctica común el tomar la etapa inicial de un compilador y rehacer su etapa final asociada para producir un compilador para el mismo lenguaje fuente en una máquina distinta. También resulta tentador crear compiladores para varios lenguajes distintos y generar el mismo lenguaje intermedio para, por último, usar una etapa final común para todos ellos, y obtener así varios compiladores para una máquina. Para ilustrar esta práctica y su inversa, la figura 1.8

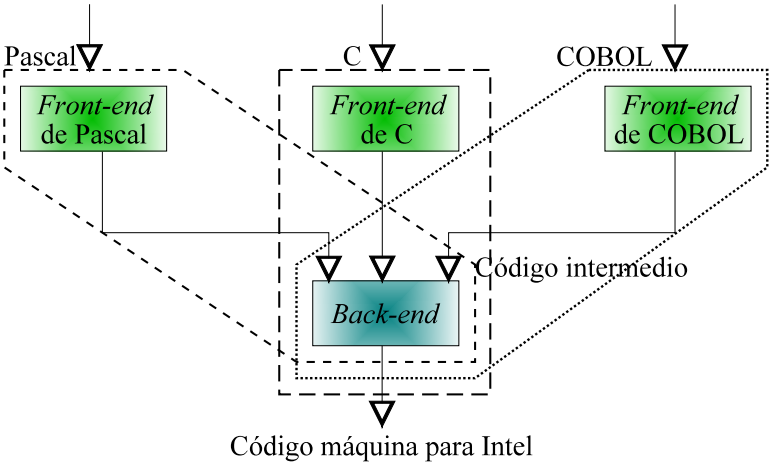


Figura 1.9. Creación de tres compiladores (Pascal, C y COBOL) para una misma máquina Intel

Muestra una situación en la que se quiere crear un compilador del lenguaje C para tres máquinas diferentes: Dec-Alpha (Unix), Motorola (Mac OS) e Intel (MS-DOS). Cada bloque de líneas punteadas agrupa un *front-end* con un *back-end* dando lugar a un compilador completo.

De manera inversa se podrían construir tres compiladores de Pascal, C y COBOL para una misma máquina, p.ej. Intel, como se ilustra en la figura 1.9.

Por último, la creación de compiladores de Pascal, C y COBOL para las máquinas Dec-Alpha, Motorola e Intel, pasaría por la combinación de los métodos anteriores, tal como ilustra la figura 1.10.

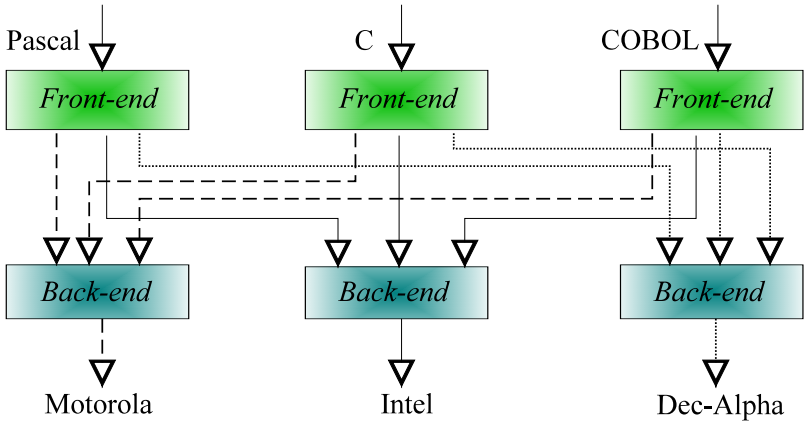


Figura 1.10. La combinación de cada *front-end* con un *back-end* da lugar a un compilador distinto: tres de Pascal, tres de C y tres de COBOL. El esfuerzo se ha reducido considerablemente.

1.3.2 La tabla de símbolos

Una función esencial de un compilador es registrar los identificadores de usuario (nombres de variables, de funciones, de tipos, etc.) utilizados en el programa fuente y reunir información sobre los distintos atributos de cada identificador. Estos atributos pueden proporcionar información sobre la memoria asignada a un identificador, la dirección de memoria en que se almacenará en tiempo de ejecución, su tipo, su ámbito (la parte del programa donde es visible), etc.

Pues bien, la tabla de símbolos es una estructura de datos que posee información sobre los identificadores definidos por el usuario, ya sean constantes, variables, tipos u otros. Dado que puede contener información de diversa índole, debe hacerse de forma que su estructura no sea uniforme, esto es, no se guarda la misma información sobre una variable del programa que sobre un tipo definido por el usuario. Hace funciones de diccionario de datos y su estructura puede ser una tabla hash, un árbol binario de búsqueda, etc., con tal de que las operaciones de acceso sean lo

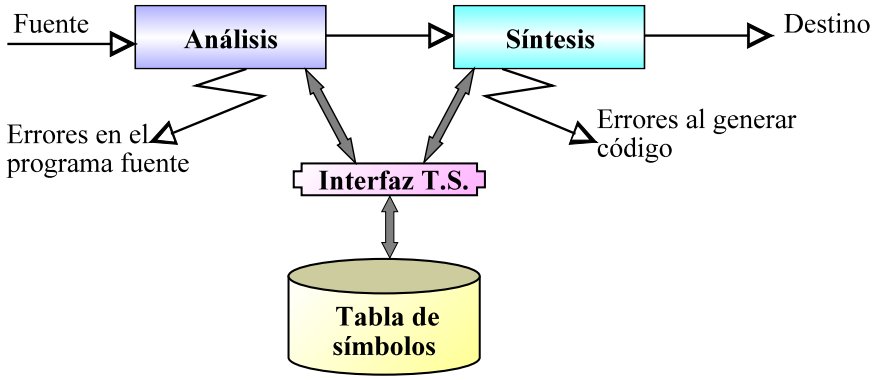


Figura 1.11. Esquema por etapas definitivo de un traductor

bastante eficientes.

Tanto la etapa de análisis como la de síntesis accede a esta estructura, por lo que se halla muy acoplada al resto de fases del compilador. Por ello conviene dotar a la tabla de símbolos de una interfaz lo suficientemente genérica como para permitir el cambio de las estructuras internas de almacenamiento sin que estas fases deban ser retocadas. Esto es así porque suele ser usual hacer un primer prototipo de un compilador con una tabla de símbolos fácil de construir (y por tanto, ineficiente), y cuando el compilador ya ha sido finalizado, entonces se procede a sustituir la tabla de símbolos por otra más eficiente en función de las necesidades que hayan ido surgiendo a lo largo de la etapa de desarrollo anterior. Siguiendo este criterio, el esquema general definitivo de un traductor se detalla en la figura 1.11. La figura 1.12 ilustra el esquema por fases, donde cada etapa ha sido sustituida por las fases que la componen y se ha hecho mención explícita del preprocesador..

1.4 Ejemplo de compilación

Vamos a estudiar a continuación las diferentes tareas que lleva a cabo cada fase de un compilador hasta llegar a generar el código asociado a una sentencia de C. La sentencia con la que se va a trabajar es la siguiente:

```
#define PORCENTAJE 8
comision = fijo + valor * PORCENTAJE;
```

Para no complicar demasiado el ejemplo, asumiremos que las variables referenciadas han sido previamente declaradas de tipo **int**, e inicializadas a los valores deseados.

Comenzaremos con el preprocesamiento hasta llegar, finalmente, a la fase de generación de código máquina en un microprocesador cualquiera.

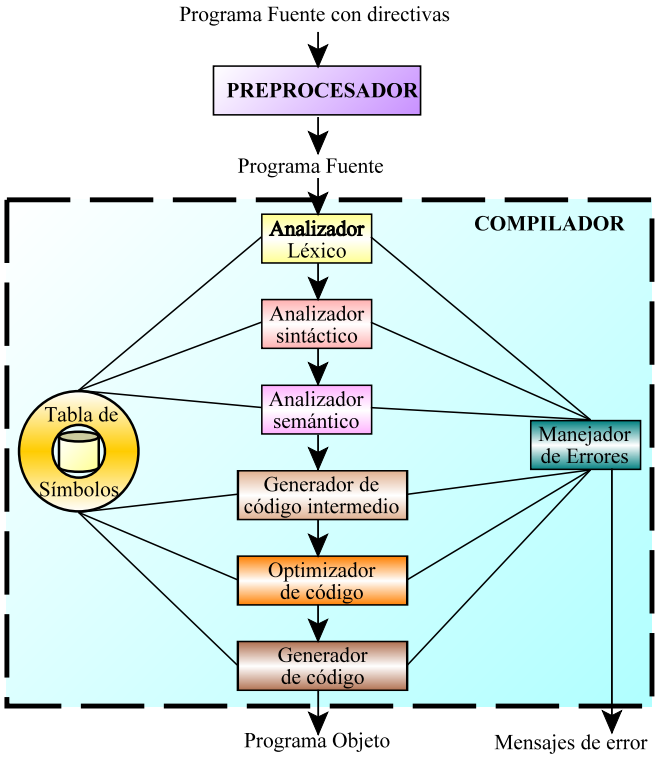


Figura 1.12. Esquema completo de un compilador por fases con preprocesador.

1.4.1 Preprocesamiento

Como ya hemos visto, el código fuente de una aplicación se puede dividir en módulos almacenados en archivos distintos. La tarea de reunir el programa fuente a menudo se confía a un programa distinto, llamado preprocesador. El preprocesador también puede expandir abreviaturas, llamadas macros, a proposiciones del lenguaje fuente. En nuestro ejemplo, la constante PORCENTAJE se sustituye por su valor, dando lugar al texto:

```
comision = fijo + valor * 8;
```

que pasa a ser la fuente que entrará al compilador.

1.4.2 Etapa de análisis

En esta etapa se controla que el texto fuente sea correcto en todos los sentidos, y se generan las estructuras necesarias para la generación de código.

1.4.2.1 Fase de análisis lexicográfico

En esta fase, la cadena de caracteres que constituye el programa fuente se lee de izquierda a derecha y se agrupa en componentes léxicos, que son secuencias de caracteres que tienen un significado atómico; además el analizador léxico trabaja con la tabla de símbolos introduciendo en ésta los nombres de las variables.

En nuestro ejemplo los caracteres de la proposición de asignación

comision= fijo + valor * 8 ;

se agruparían en los componentes léxicos siguientes:

1.- El identificador comision.
2.- El símbolo de asignación '='.
3.- El identificador fijo.
4.- El signo de suma '+'.
5.- El identificador valor.
6.- El signo de multiplicación '*'.
7.- El número 8.
8.- El símbolo de fin de sentencia ';'.

La figura 1.13 ilustra cómo cada componente léxico se traduce a su categoría gramatical, y se le asocia alguna información, que puede ser un puntero a la tabla de símbolos donde se describe el identificador, o incluso un valor directo, como ocurre en el caso del literal 8. Así, cada componente se convierte en un par (categoría, atributo), y se actualiza la tabla de símbolos. Esta secuencia de pares se le pasa a la siguiente fase de análisis.

Nótese como los espacios en blanco que separan los caracteres de estos componentes léxicos normalmente se eliminan durante el análisis léxico, siempre y cuando la definición del lenguaje a compilar así lo aconseje, como ocurre en C. Lo mismo pasa con los tabuladores innecesarios y con los retornos de carro. Los comentarios, ya estén anidados o no, también son eliminados.

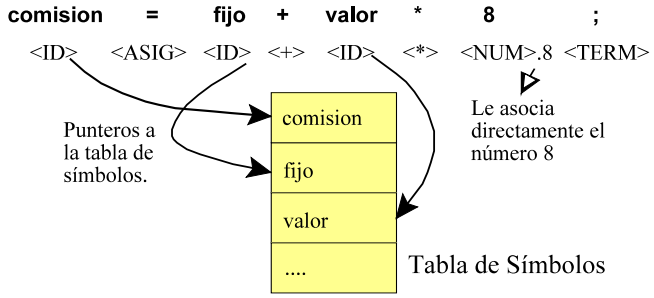


Figura 1.13. Transformación realizada por el analizador lexicográfico

1.4.2.2 Fase de análisis sintáctico

Trabaja con una gramática de contexto libre y genera el árbol sintáctico que

reconoce su sentencia de entrada. En nuestro caso las categorías gramaticales del análisis léxico son los terminales de la gramática. Para el ejemplo que nos ocupa podemos partir de la gramática:

S → <ID> <ASIG> expr <TERM>
expr → <ID>
| <ID> <+> expr
| <ID> <*> expr
| <NUM>

de manera que el análisis sintáctico intenta generar un árbol sintáctico que encaje con la sentencia de entrada. Para nuestro ejemplo, dicho árbol sintáctico existe y es el de la figura 1.14. El árbol puede representarse tal y como aparece en esta figura, o bien invertido.

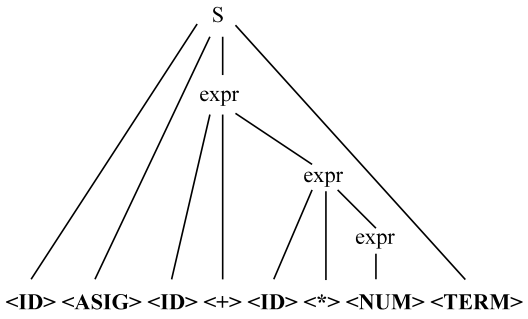


Figura 1.14. Árbol sintáctico de la sentencia de entrada. Aunque sólo se han representado las categorías gramaticales, recuérdese que cada una lleva o puede llevar asociado un atributo.

1.4.2.2.1 Compilación dirigida por sintaxis

Se ha representado una situación ideal en la que la fase lexicográfica actúa por separado y, sólo una vez que ha acabado, le suministra la sintáctica su resultado de salida. Aunque proseguiremos en nuestro ejemplo con esta clara distinción entre fases, es importante destacar que el analizador sintáctico tiene el control en todo momento, y el léxico por trozos, a petición del sintáctico. En otras palabras, el sintáctico va construyendo su árbol poco a poco (de izquierda a derecha), y cada vez que necesita un nuevo componente léxico para continuar dicha construcción, se lo solicita al lexicográfico; éste lee nuevos caracteres del fichero de entrada hasta conformar un nuevo componente y, una vez obtenido, se lo suministra al sintáctico, quien continúa la construcción del árbol hasta que vuelve a necesitar otro componente, momento en que se reinicia el proceso. Este mecanismo finaliza cuando se ha obtenido el árbol y ya

no hay más componentes en el fichero de entrada, o bien cuando es imposible construir el árbol.

Esto es tan sólo el principio de lo que se denomina **compilación dirigida por sintaxis** (ver figura 1.15): es aquél mecanismo de compilación en el que el control lo lleva el analizador sintáctico, y todas las demás fases están sometidas a él.

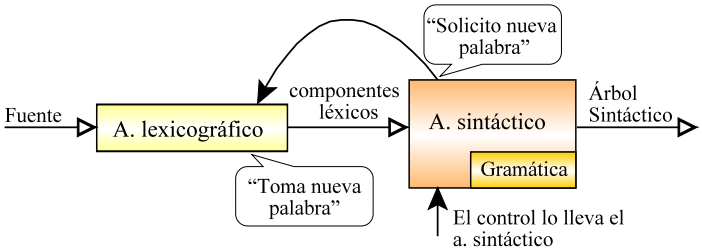


Figura 1.15. Análisis dirigido por sintaxis

1.4.2.3 Fase de análisis semántico

Esta fase revisa el árbol sintáctico junto con los atributos y la tabla de símbolos para tratar de encontrar errores semánticos. Para todo esto se analizan los operadores y operandos de expresiones y proposiciones. Finalmente reúne la información necesaria sobre los tipos de datos para la fase posterior de generación de código.

El componente más importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si los operandos de cada operador son compatibles según la especificación del lenguaje fuente. Si suponemos que nuestro lenguaje solo trabaja con números reales, la salida de esta fase sería su mismo árbol, excepto porque el atributo de <NUM>, que era el entero 8 a la entrada, ahora pasaría a ser el real 8,0. Además se ha debido controlar que las variables implicadas en la sentencia, a saber, comision, fijo y valor son compatibles con el tipo numérico de la constante 8,0.

1.4.3 Etapa de síntesis

En la etapa anterior se ha controlado que el programa de entrada es correcto. Por tanto, el compilador ya se encuentra en disposición de generar el código máquina equivalente semánticamente al programa fuente. Para ello se parte de las estructuras generadas en dicha etapa anterior: árbol sintáctico y tabla de símbolos.

1.4.3.1 Fase de generación de código intermedio

Después de la etapa de análisis, se suele generar una representación intermedia explícita del programa fuente. Dicha representación intermedia se puede considerar como un programa para una máquina abstracta.

Cualquier representación intermedia debe tener dos propiedades importantes; debe ser fácil de generar y fácil de traducir al código máquina destino. Así, una representación intermedia puede tener diversas formas. En el presente ejemplo se trabajará con una forma intermedia llamada "código de tres direcciones", que es muy parecida a un lenguaje ensamblador para un microprocesador que carece de registros y sólo es capaz de trabajar con direcciones de memoria y literales. En el código de tres direcciones cada instrucción tiene como máximo tres operandos. Siguiendo el ejemplo propuesto, se generaría el siguiente código de tres direcciones:

```
t1 = 8.0
t2 = valor * t1
t3 = fijo + t2
comision = t3
```

De este ejemplo se pueden destacar varias propiedades del código intermedio escogido:

- Cada instrucción de tres direcciones tiene a lo sumo un operador, además de la asignación.
- El compilador debe generar un nombre temporal para guardar los valores intermedios calculados por cada instrucción: t1, t2 y t3.
- Algunas instrucciones tienen menos de tres operandos, como la primera y la última instrucciones del ejemplo.

1.4.3.2 Fase de optimización de código

Esta fase trata de mejorar el código intermedio, de modo que en la siguiente fase resulte un código de máquina más rápido de ejecutar. Algunas optimizaciones son triviales. En nuestro ejemplo hay una forma mejor de realizar el cálculo de la comisión, y pasa por realizar sustituciones triviales en la segunda y cuarta instrucciones, obteniéndose:

```
t2 = valor * 8.0
comision= fijo + t2
```

El compilador puede deducir que todas las apariciones de la variable t1 pueden sustituirse por la constante 8,0, ya que a t1 se le asigna un valor que ya no cambia, de modo que la primera instrucción se puede eliminar. Algo parecido sucede con la variable t3, que se utiliza sólo una vez, para transmitir su valor a comision en una asignación directa, luego resulta seguro sustituir comision por t3, a raíz de lo cual se elimina otra de las líneas del código intermedio.

1.4.3.3 Fase de generación de código máquina

La fase final de un compilador es la generación de código objeto, que por lo general consiste en código máquina reubicable o código ensamblador. Cada una de las variables usadas por el programa se traduce a una dirección de memoria (esto también se ha podido hacer en la fase de generación de código intermedio). Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a

registros.

Siguiendo el mismo ejemplo, y utilizando los registros R1 y R2 de un microprocesador hipotético, la traducción del código optimizado podría ser:

```
MOVE [1Ah], R1
MULT #8.0, R1
MOVE [15h], R2
ADD R1, R2
MOVE R2, [10h]
```

El primer y segundo operandos de cada instrucción especifican una fuente y un destino, respectivamente. Este código traslada el contendido de la dirección [1Ah] al registro R1, después lo multiplica por la constante real 8.0. La tercera instrucción pasa el contenido de la dirección [15h] al registro R2. La cuarta instrucción le suma el valor previamente calculado en el registro R1. Por último el valor del registro R2 se pasa a la dirección [10h]. Como el lector puede suponer, la variable comision se almacena en la dirección [10h], fijo en [15h] y valor en [1Ah].

Capítulo 2

Análisis lexicográfico

2.1 Visión general

Este capítulo estudia la primera fase de un compilador, es decir su análisis lexicográfico, también denominado abreviadamente análisis léxico. Las técnicas utilizadas para construir analizadores léxicos también se pueden aplicar a otras áreas como, por ejemplo, a lenguajes de consulta y sistemas de recuperación de información. En cada aplicación, el problema de fondo es la especificación y diseño de programas que ejecuten las acciones activadas por palabras que siguen ciertos patrones dentro de las cadenas a reconocer. Como la programación dirigida por patrones está ampliamente extendida y resulta de indudable utilidad, existen numerosos metalenguajes que permiten establecer pares de la forma patrón-acción, de manera que la acción se ejecuta cada vez que el sistema se encuentra una serie de caracteres cuya estructura coincide con la del patrón. En concreto, vamos a estudiar Lex con el objetivo de especificar los analizadores léxicos. En este lenguaje, los patrones se especifican por medio de expresiones regulares, y el metacompilador de Lex genera un reconocedor de las expresiones regulares mediante un autómata finito (determinista evidentemente) eficiente.

Por otro lado, una herramienta software que automatiza la construcción de analizadores léxicos permite que personas con diferentes conocimientos utilicen la concordancia de patrones en sus propias áreas de aplicación, ya que, a la hora de la verdad, no es necesario tener profundos conocimientos de informática para aplicar dichas herramientas.

2.2 Concepto de analizador léxico

Se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones.

La entrada del analizador léxico podemos definirla como una secuencia de caracteres, que pueda hallarse codificada según cualquier estándar: ASCII (*American Standard Code for Information Interchange*), EBCDIC (*Extended Binary Coded Decimal Interchange Code*), Unicode, etc. El analizador léxico divide esta secuencia en palabras con significado propio y después las convierte a una secuencia de terminales desde el punto de vista del analizador sintáctico. Dicha secuencia es el punto de partida para que el analizador sintáctico construya el árbol sintáctico que reconoce la/s sentencia/s de entrada, tal y como puede verse en la figura 2.1.

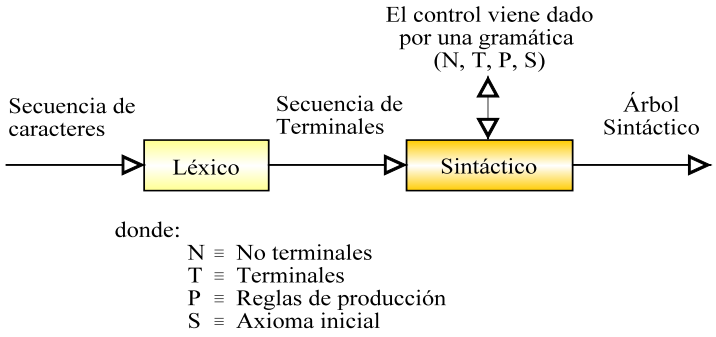


Figura 2.1. Entradas y salidas de las dos primeras fases de la etapa de análisis. La frase "Secuencia de Terminales" hace referencia a la gramática del sintáctico; pero también es posible considerar que dicha secuencia es de no terminales si usamos el punto de vista del lexicográfico.

El analizador léxico reconoce las palabras en función de una gramática regular de manera que el alfabeto Σ de dicha gramática son los distintos caracteres del juego de caracteres del ordenador sobre el que se trabaja (que forman el conjunto de símbolos terminales), mientras que sus no terminales son las categorías léxicas en que se integran las distintas secuencias de caracteres. Cada no terminal o categoría léxica de la gramática regular del análisis léxico es considerado como un terminal de la gramática de contexto libre con la que trabaja el analizador sintáctico, de manera que la salida de alto nivel (no terminales) de la fase léxica supone la entrada de bajo nivel (terminales) de la fase sintáctica. En el caso de Lex, por ejemplo, la gramática regular se expresa mediante expresiones regulares.

2.2.1 Funciones del analizador léxico

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta

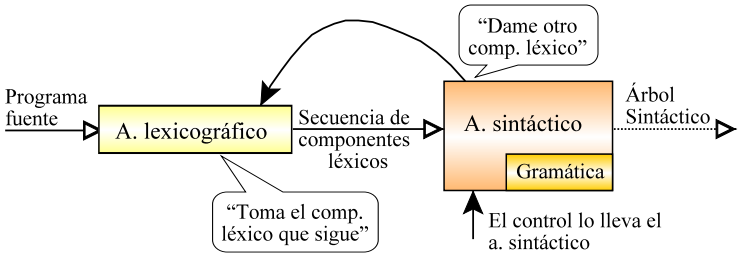


Figura 2.2. La fase de análisis léxico se halla bajo el control del análisis sintáctico. Normalmente se implementa como una función de éste

interacción suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden “Dame el siguiente componente léxico”del analizador sintáctico, el léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico, el cual devuelve al sintáctico según el formato convenido (ver figura 2.2).

Además de esta función principal, el analizador léxico también realiza otras de gran importancia, a saber:

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc, y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje, etc., y tratarlos correctamente con respecto a la tabla de símbolos (solo en los casos en que este analizador deba tratar con dicha estructura).
- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información acerca de dónde se ha producido.
- Avisar de errores léxicos. Por ejemplo, si el carácter ‘@’ no pertenece al lenguaje, se debe emitir un error.
- También puede hacer funciones de preprocesador.

2.2.2 Necesidad del analizador léxico

Un buen profesional debe ser capaz de cuestionar y plantearse todas las decisiones de diseño que se tomen, y un asunto importante es el porqué se separa el análisis léxico del sintáctico si, al fin y al cabo, el control lo va a llevar el segundo. En otras palabras, por qué no se delega todo el procesamiento del programa fuente sólo en el análisis sintáctico, cosa perfectamente posible (aunque no plausible como veremos a continuación), ya que el sintáctico trabaja con gramáticas de contexto libre y éstas engloban a la regulares. A continuación estudiaremos algunas razones de esta separación.

2.2.2.1 Simplificación del diseño

Un diseño sencillo es quizás la ventaja más importante. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una, otra o ambas fases. Normalmente añadir un analizador léxico permite simplificar notablemente el analizador sintáctico. Aún más, la simplificación obtenida se hace especialmente patente cuando es necesario realizar modificaciones o extensiones al lenguaje inicialmente ideado; en otras palabras, se facilita el mantenimiento del compilador a medida que el lenguaje evoluciona.

La figura 2.3 ilustra una situación en la que, mediante los patrones correspondientes, el analizador léxico reconoce números enteros y operadores aritméticos. A la hora de construir una primera versión del analizador sintáctico,

podemos asumir que dos expresiones pueden ir conectadas con cualquiera de dichos operadores, por lo que se puede optar por agruparlos todos bajo la categoría léxica **OPARIT** (Operadores ARITméticos). En una fase posterior, puede resultar necesario disgregar dicha categoría en tantas otras como operadores semánticamente diferentes haya: **OPARIT** desaparece y aparecen **MAS**, **MENOS**, **MULT** y **DIV**. Una modificación tal resulta trivial si se han separado adecuadamente ambos analizadores, ya que consiste en sustituir el patrón agrupado ('-'|'+'|'*'|'/') por los patrones disgregados '-', '+', '*' y '/'.

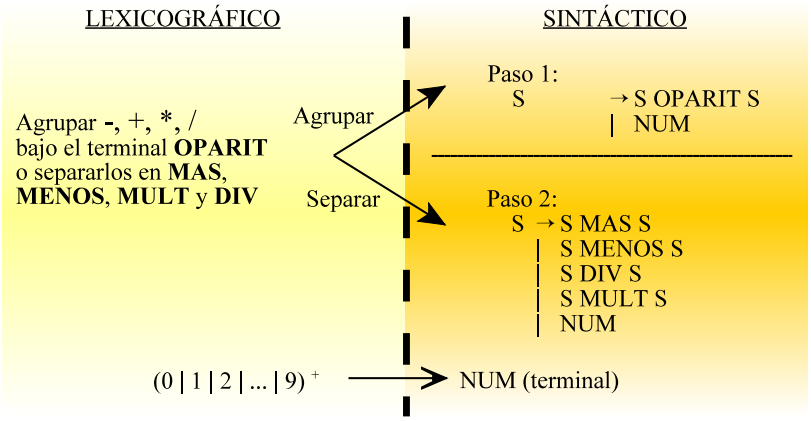


Figura 2.3. Pasos en la construcción progresiva de un compilador

Si el sintáctico tuviera la gramática del paso 1, el lexicográfico sería:

el patrón (0 | 1 | 2 | ... | 9) $^{+}$ retorna la categoría **NUM**
el patrón ('+' | '-' | '*' | '/') retorna la categoría **OPARIT**

En cambio, si el sintáctico adopta el paso 2, el lexicográfico sería:

el patrón (0 | 1 | 2 | ... | 9) $^{+}$ retorna la categoría **NUM**
el patrón '+' retorna la categoría **MAS**
el patrón '-' retorna la categoría **MENOS**
el patrón '*' retorna la categoría **MULT**
el patrón '/' retorna la categoría **DIV**

También se puede pensar en eliminar el lexicográfico, incorporando su gramática en la del sintáctico, de manera que éste vería incrementado su número de reglas con las siguientes:

NUM → 0
| 1
| 2
| 3
...

| NUM NUM

sin embargo, los autómatas destinados a reconocer los componentes léxicos y al árbol sintáctico son radicalmente diferentes, tanto en su concepción como en su implementación lo que, de nuevo, nos lleva a establecer una división entre estos análisis.

A modo de conclusión, se puede decir que es muy recomendable trabajar con dos gramáticas, una que se encarga del análisis léxico y otra que se encarga del análisis sintáctico. ¿Dónde se pone el límite entre lo que reconoce una y otra gramática?, ¿qué se considera un componente básico?, ¿cuántas categorías gramaticales se establecen? Si se crean muchas categorías gramaticales se estará complicando la gramática del sintáctico, como ocurre p.ej. en el paso 2. En general, deben seguirse un par de reglas básicas para mantener la complejidad en unos niveles admisibles. La primera es que la información asociada a cada categoría léxica debe ser la necesaria y suficiente, lo que quedará más claro en capítulos posteriores, cuando se conozca el concepto de **atributo**. La segunda es que, por regla general, las gramáticas que se planteen (regular y de contexto libre) no deben verse forzadas, en el sentido de que los distintos conceptos que componen el lenguaje de programación a compilar deben formar parte de una o de otra de forma natural; p.ej., el reconocimiento que deba hacerse carácter a carácter (sin que éstos tengan un significado semántico por sí solos) debe formar parte del análisis léxico.

2.2.2.2 Eficiencia

La división entre análisis léxico y sintáctico también mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para las funciones explicadas en el epígrafe 2.2.1. Gran parte del tiempo de compilación se invierte en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de *buffers* para la lectura de caracteres de entrada y procesamiento de patrones se puede mejorar significativamente el rendimiento de un compilador.

2.2.2.3 Portabilidad

Se mejora la portabilidad del compilador, ya que las peculiaridades del alfabeto de partida, del juego de caracteres base y otras anomalías propias de los dispositivos de entrada pueden limitarse al analizador léxico. La representación de símbolos especiales o no estándares, como '↑' en Pascal, se pueden aislar en el analizador léxico.

Por otro lado, algunos lenguajes, como APL (A Program Language) se benefician sobremanera del tratamiento de los caracteres que forman el programa de entrada mediante un analizador aislado. El Diccionario de Informática de la *Oxford University Press* define este lenguaje de la siguiente forma:

«*... Su característica principal es que proporciona un conjunto muy grande de operadores importantes para tratar las órdenes multidimensionales junto con la*

capacidad del usuario de definir sus propios operadores. Los operadores incorporados se encuentran representados, principalmente, por caracteres solos que utilizan un conjunto de caracteres especiales. De este modo, los programas APL son muy concisos y, con frecuencia, impenetrables».

2.2.2.4 Patrones complejos

Otra razón por la que se separan los dos análisis es para que el analizador léxico se centre en el reconocimiento de componentes básicos complejos. Por ejemplo en Fortran, existe el siguiente par de proposiciones muy similares sintácticamente, pero de significado bien distinto:

```
DO5I = 2.5        (Asignación del valor 2.5 a la variable DO5I)
DO 5 I = 2, 5     (Bucle que se repite para I = 2, 3, 4 y 5)
```

En este lenguaje los espacios en blancos no son significativos fuera de los comentarios y de un cierto tipo de cadenas (para ahorrar espacio de almacenamiento, en una época de la Informática en la que éste era un bien escaso), de modo que supóngase que todos los espacios en blanco eliminables se suprimen antes de comenzar el análisis léxico. En tal caso, las proposiciones anteriores aparecerían ante el analizador léxico como:

```
DO5I=2.5
DO5I=2,5
```

El analizador léxico no sabe si DO es una palabra reservada o es el prefijo del nombre de una variable hasta que se lee la coma. Ha sido necesario examinar la cadena de entrada mucho más allá de la propia palabra a reconocer haciendo lo que se denomina *lookahead* (o prebúsqueda). La complejidad de este procesamiento hace recomendable aislarlo en una fase independiente del análisis sintáctico.

En cualquier caso, en lenguajes como Fortran primero se diseñó el lenguaje y luego el compilador, lo que conllevó problemas como el que se acaba de plantear. Hoy día los lenguajes se diseñan teniendo en mente las herramientas de que se dispone para la construcción de su compilador y se evitan este tipo de situaciones.

2.3 *Token*, patrón y lexema

Desde un punto de vista muy general, podemos abstraer el programa que implementa un análisis léxicográfico mediante una estructura como:

```
(Expresión regular)1      {acción a ejecutar}1

(Expresión regular)2      {acción a ejecutar}2
(Expresión regular)3      {acción a ejecutar}3
...                       ...
(Expresión regular)n      {acción a ejecutar}n
```

donde cada acción a ejecutar es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando la secuencia de entrada coincida con la expresión regular. Normalmente esta acción suele finalizar con la devolución de una

categoría léxica.

Todo esto nos lleva a los siguientes conceptos de fundamental importancia a lo largo de nuestro estudio:

- ✎ Patrón: es una expresión regular.
- ✎ *Token*: es la categoría léxica asociada a un patrón. Cada *token* se convierte en un número o código identificador único. En algunos casos, cada número tiene asociada información adicional necesaria para las fases posteriores de la etapa de análisis. El concepto de *token* coincide directamente con el concepto de terminal desde el punto de vista de la gramática utilizada por el analizador sintáctico.
- ✎ Lexema: Es cada secuencia de caracteres concreta que encaja con un patrón. P.ej: "8", "23" y "50" son algunos lexemas que encajan con el patrón ('0'|'1'|'2'| ... |'9')$^+$. El número de lexemas que puede encajar con un patrón puede ser finito o infinito, p.ej. en el patrón 'W''H''I''L''E' sólo encaja el lexema "WHILE".

Una vez detectado que un grupo de caracteres coincide con un patrón, se considera que se ha detectado un lexema. A continuación se le asocia el número de su categoría léxica, y dicho número o *token* se le pasa al sintáctico junto con información adicional, si fuera necesario. En la figura 1.13 puede verse cómo determinadas categorías llevan información asociada, y otras no.

Por ejemplo, si se necesita construir un analizador léxico que reconozca los números enteros, los números reales y los identificadores de usuario en minúsculas, se puede proponer una estructura como:

Expresión Regular	Terminal asociado
(0 ... 9)$^+$	NUM_ENT
(0 ... 9)*. (0 ... 9)$^+$	NUM_REAL
(a ... z) (a ... z 0 ... 9)*	ID

Asociado a la categoría gramatical de número entero se tiene el *token* **NUM_ENT** que puede equivaler, p.ej. al número 280; asociado a la categoría gramatical número real se tiene el *token* **NUM_REAL** que equivale al número 281; y la categoría gramatical identificador de usuario tiene el *token* ID que equivale al número 282. Así, la estructura expresión regular-acción sería la siguiente (supuesto que las acciones las expresamos en C o Java):

(0 ... 9)$^+$	{ return 280;}
(0 ... 9)*. (0 ... 9)$^+$	{ return 281;}
(a ... z) (a ... z 0...9)*	{ return 282;}
" "	{ }

De esta manera, un analizador léxico que obedeciera a esta estructura, si durante su ejecución se encuentra con la cadena:

95.7 99 cont

intentará leer el lexema más grande de forma que, aunque el texto “95" encaja con el primer patrón, el punto y los dígitos que le siguen “.7" hacen que el analizador decida reconocer “95.7" como un todo, en lugar de reconocer de manera independiente “95" por un lado y “.7" por otro; así se retorna el *token* **NUM_REAL**. Resulta evidente que un comportamiento distinto al expuesto sería una fuente de problemas. A continuación el patrón “ ” y la acción asociada permiten ignorar los espacios en blanco. El “99" coincide con el patrón de **NUM_ENT**, y la palabra “cont” con **ID**.

Para facilitar la comprensión de las acciones asociadas a los patrones, en vez de trabajar con los números 280, 281 y 282 se definen mnemotécnicos.

```
# define NUM_ENT 280
# define NUM_REAL 281
# define ID 282
(“ ”\t \n)
(0 ... 9)+                    {return NUM_ENT;}
(0 ... 9)*. (0 ... 9)+        {return NUM_REAL;}
(a ... z) (a ... z 0 ... 9)*  {return ID;}
```

En esta nueva versión, los lexemas que entran por el patrón (“ ”\t \n) no tienen acción asociada, por lo que, por defecto, se ejecuta la acción nula o vacía.

El asociar un número (*token*) a cada categoría gramatical se suele emplear mucho en los metacompiladores basados en lenguajes puramente imperativos, como pueda ser C o Pascal. Los metacompiladores más modernos basados en programación orientada a objetos también asocian un código a cada categoría gramatical, pero en lugar de retornar dicho código, retornan objetos con una estructura más compleja.

2.3.1 Aproximaciones para construir un analizador lexicográfico

Hay tres mecanismos básicos para construir un analizador lexicográfico:

- Ad hoc. Consiste en la codificación de un programa reconocedor que no sigue los formalismos propios de la teoría de autómatas. Este tipo de construcciones es muy propensa a errores y difícil de mantener.
- Mediante la implementación manual de los autómatas finitos. Este mecanismo consiste en construir los patrones necesarios para cada categoría léxica, construir sus automátas finitos individuales, fusionarlos por opcionalidad y, finalmente, implementar los autómatas resultantes. Aunque la construcción de analizadores mediante este método es sistemática y no propensa a errores, cualquier actualización de los patrones reconocedores implica la modificación del código que los implementa, por lo que el mantenimiento se hace muy costoso.
- Mediante un metacompilador. En este caso, se utiliza un programa especial que tiene como entrada pares de la forma (expresión regular, acción). El

metacompilador genera todos los autómatas finitos, los convierte a autómata finito determinista, y lo implementa en C. El programa C así generado se compila y se genera un ejecutable que es el análizador léxico de nuestro lenguaje. Por supuesto, existen metacompiladores que generan código Java, Pascal, etc. en lugar de C.

Dado que hoy día existen numerosas herramientas para construir analizadores léxicos a partir de notaciones de propósito especial basadas en expresiones regulares, nos basaremos en ellas para proseguir nuestro estudio dado que, además, los analizadores resultantes suelen ser bastante eficientes tanto en tiempo como en memoria. Comenzaremos con un generador de analizadores léxicos escritos en C, y continuaremos con otro que genera código Java.

2.4 El generador de analizadores lexicográficos: PCLex

En esta sección se describe la herramienta actualmente más extendida, llamada Lex, para la especificación de analizadores léxicos en general, aunque en nuestro caso nos centraremos en el analizador léxico de un compilador. La herramienta equivalente para entorno DOS se denomina PCLex, y la especificación de su entrada, lenguaje Lex. El estudio de una herramienta existente permitirá mostrar cómo, utilizando expresiones regulares, se puede combinar la especificación de patrones con acciones, para p.ej., realizar inserciones en una tabla de símbolos.

2.4.1 Visión general

Como ya sabemos, las reglas de reconocimiento son de la forma:

p_1 {acción$_1$}
p_2 {acción$_2$}
... ...
p_n {acción$_n$}

donde p_i es una expresión regular y acción$_i$ es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando se encuentra con un lexema que encaja por p_i. En Lex, las acciones se escriben en C, aunque existen multitud de metacompiladores similares al PCLex que permiten codificar en otros lenguajes, como por ejemplo Jflex que utiliza el lenguaje Java.

Un analizador léxico creado por PCLex está diseñado para comportarse en sincronía con un analizador sintáctico. Para ello, entre el código generado existe una función llamada **yylex()** que, al ser invocada (normalmente por el sintáctico), comienza a leer la entrada, carácter a carácter, hasta que encuentra el mayor prefijo en la entrada que concuerde con una de las expresiones regulares p_i; dicho prefijo constituye un lexema y, una vez leído, se dice que "ha sido consumido" en el sentido de que el punto de lectura ha avanzado hasta el primer carácter que le sigue. A continuación **yylex()** ejecuta la acción$_i$. Generalmente esta acción$_i$ devolverá el control al analizador sintáctico informándole del *token* encontrado. Sin embargo, si la acción no tiene un

return, el analizador léxico se dispondrá a encontrar un nuevo lexema, y así sucesivamente hasta que una acción devuelva el control al analizador sintáctico, o hasta llegar al final de la cadena, representada por el carácter EOF (*End Of File*). La búsqueda repetida de lexemas hasta encontrar una instrucción **return** explícita permite al analizador léxico procesar espacios en blanco y comentarios de manera apropiada.

Antes de ejecutar la acción asociada a un patrón, **yylex()** almacena el lexema leído en la variable **yytext**. Cualquier información adicional que se quiera comunicar a la función llamante (normalmente el analizador sintáctico), además del *token* debe almacenarse en la variable global **yylval**, cuyo tipo se verá más adelante.

Los programas que se obtienen con PCLex son relativamente grandes, (aunque muy rápidos también), aunque esto no suele ser un problema ante las enormes cantidades de memoria que se manejan hoy día. La gran ventaja de PCLex es que permite hacer analizadores complejos con bastante rapidez.

2.4.2 Creación de un analizador léxico

Como ya se ha comentado, PCLex tiene su propio lenguaje, al que llamaremos Lex y que permite especificar la estructura abstracta de un analizador léxico, tanto en lo que respecta a las expresiones regulares como a la acción a tomar al encontrar un lexema que encaje en cada una de ellas.

Los pasos para crear un analizador léxico con esta herramienta son (figura 2.4):

- Construir un fichero de texto en lenguaje Lex que contiene la estructura abstracta del analizador.
- Metacompilar el fichero anterior con PCLex. Así se obtendrá un fichero fuente en C estándar. Algunas veces hay que efectuar modificaciones directas en este código, aunque las últimas versiones de PCLex han disminuido al máximo estas situaciones.
- Compilar el fuente en C generado por PCLex con un compilador C, con lo que obtendremos un ejecutable que sigue los pasos descritos en el epígrafe 2.4.1.

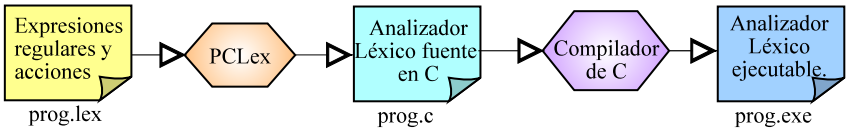


Figura 2.4. Obtención de un programa ejecutable a partir de una especificación en Lex

Si ejecutamos **prog.exe** la aplicación se quedará esperando a que se le introduzca la cadena de entrada por teclado para proceder a su partición en lexemas; el fin de fichero se introduce pulsando las teclas Ctrl y Z (en pantalla aparece **^Z**). Así el programa parte la cadena de entrada en los lexemas más largos posible, y para cada una de ellos ejecuta la acción asociada al patrón por el que encaje. Por ejemplo, si el

fuente en Lex es de la forma:

```
[0-9] + {printf("numero");}
[A-Z]+ {printf("palabra");}
```

y tecleamos :

```
HOLA 23 ^z
```

cuando un lexema casa con un patrón, **yylex()** cede el control a su acción. De esta forma se obtiene por pantalla lo siguiente:

```
palabra numero
```

La palabra "HOLA" encaja por el segundo patrón y la palabra "23" encaja por el primero. Nótese que no hemos especificado ningún patrón sobre los espacios en blanco; por defecto, la función **yylex()** generada cuando se encuentra un carácter o secuencia de caracteres que no casa con ningún patrón, sencillamente lo visualiza por la salida estándar y continúa a reconocer el siguiente lexema.

Para no tener que introducir la cadena fuente por teclado, sino que sea reconocida desde un fichero, podemos, o bien redirigir la entrada con:

```
prog < file.pas > salida.txt
```

donde:

```
< file.pas        redirige la entrada y
> salida.txt      redirige la salida,
```

o bien hacer uso de las variables **yyin** e **yyout** de tipo ***FILE** para inicializarlas a los ficheros correspondientes. Estas variables son suministradas por el código generado por PCLex, e inicialmente apuntan a **stdin** y **stdout** respectivamente.

2.4.3 El lenguaje Lex

Un programa Lex tiene la siguiente estructura:

```
Área de definiciones Lex
%%              /* es lo único obligatorio en todo el programa */
Área de reglas
%%
Área de funciones
```

siendo el mínimo programa que se puede construir en Lex:

```
%%
```

En el área de reglas se definen los patrones de los lexemas que se quieren buscar a la entrada, y al lado de tales expresiones regulares, se detallan (en C) las acciones a ejecutar tras encontrar una cadena que se adapte al patrón indicado. Los separadores "%%" deben ir obligatoriamente en la columna 0, al igual que las reglas y demás información propia del lenguaje Lex.

Como ya se indicó, en Lex, si una cadena de entrada no encaja con ningún patrón, la acción que se toma es escribir tal entrada en la salida. Por tanto, como el programa %% no especifica ningún patron, pues el analizador léxico que se genera lo único que hace es copiar la cadena de entrada en la salida que, por defecto, es la salida estándar.

2.4.3.1 Premisas de Lex para reconocer lexemas

El **yylex()** generado por PCLex sigue dos directrices fundamentales para reconocer lexemas en caso de ambigüedad. Estas directrices son, por orden de prioridad:

1. Entrar siempre por el patrón que reconoce el lexema más largo posible.
2. En caso de conflicto usa el patrón que aparece en primera posición.

Como consecuencia de la segunda premisa: los patrones que reconocen palabras reservadas se colocan siempre antes que el patrón de identificador de usuario. P.ej. un analizador léxico para Pascal (en el que TYPE y VAR son palabras reservadas), podría tener una apariencia como:

```
%%
"TYPE"
"VAR"
[A-Z][A-Z0-9]*
...
```

Cuando **yylex()** se encuentra con la cadena "VAR" se produce un conflicto, (ya que dicho lexema puede entrar tanto por el patrón segundo, como por el tercero). Entonces toma el patrón que aparece antes, que en el ejemplo sería "VAR", reconociéndose al lexema como palabra reservada. Cambiar el orden de ambos patrones tiene consecuencias funestas:

```
%%
"TYPE"
[A-Z][A-Z0-9]*
"VAR"
...
```

ya que esta vez el lexema "VAR" entraría por el patrón de identificador de usuario, y jamás se reconocería como una palabra reservada: el patrón "VAR" es superfluo.

2.4.3.2 Caracteres especiales de Lex

Lex se basa en el juego de caracteres ASCII para representar las expresiones regulares, por lo que los caracteres que veremos a continuación tienen un significado especial con tal propósito:

"": sirve para encerrar cualquier cadena de literales. Por regla general no es necesario encerrar los literales entre comillas a no ser que incluyan símbolos especiales, esto es, el patrón "WHILE" y el patrón WHILE son equivalentes; pero para representar p.ej. el inicio de comentario en Modula-2 sí es necesario entrecomillar los caracteres que componen al patrón: "(*", ya que éste contiene, a su vez, símbolos especiales.

\: hace literal al siguiente carácter. Ej.: \" reconoce unas comillas. También se utiliza para expresar aquellos caracteres que no tienen representación directa por pantalla: \n para el retorno de carro, \t para el tabulador, etc.

nºoctal: representa el carácter cuyo valor ASCII es **nº octal**. P.ej: \012 reconoce el carácter decimal 10 que se corresponde con LF (*Line Feed*).

[ ]: permiten especificar listas de caracteres, o sea uno de los caracteres que encierra, ej.: [abc] reconoce o la 'a', o la 'b', o la 'c', ([abc] ≡ (a|b|c)). Dentro de los corchetes los siguientes caracteres también tienen un sentido especial:

- -: indica rango. Ej.: [A-Z0-9] reconoce cualquier carácter de la 'A' a la 'Z' o del '0' a '9'.
- ^: indica compleción cuando aparece al comienzo, justo detrás de "[". Ej.: [^abc] reconoce cualquier carácter excepto la 'a', la 'b' o la 'c'. Ej.: [^A-Z] reconoce cualquier carácter excepto los de la 'A' a la 'Z'.

?: aquello que le precede es opcional[1]. Ej.: a? ≡ (a | ε). Ej.: [A-Z]? reconoce cualquier letra de la 'A' a la 'Z' o bien ε. Ej.: a?b ≡ ab | εb.

.: representa a cualquier carácter (pero sólo a uno) excepto el retorno de carro (\n). Es muy interesante porque nos permite recoger cualquier otro carácter que no sea reconocido por los patrones anteriores.

|: indica opcionalidad (*OR*). Ej.: a|b reconoce a la 'a' o a la 'b'. Ej.: .|\n reconoce cualquier carácter. Resulta curioso el patrón (.|\n)* ya que por aquí entra el programa entero, y como **yylex()** tiene la premisa de reconocer el lexema más largo, pues probablemente ignorará cualquier otro patrón. También resulta probable que durante el reconocimiento se produzca un error por desbordamiento del espacio de almacenamiento de la variable **yytext** que, recordemos, almacena el lexema actual.

*: indica repetición 0 o más veces de lo que le precede.

+: indica repetición 1 o más veces de lo que le precede.

( ): permiten la agrupación (igual que en las expresiones aritméticas).

{ }: indican rango de repetición. Ej.: a{1,5} ≡ aa?a?a?a? Las llaves vienen a ser algo parecido a un * restringido. También nos permite asignarle un nombre a una expresión regular para reutilizarla en múltiples patrones; esto se verá más adelante.

2.4.3.3 Caracteres de sensibilidad al contexto

Lex suministra ciertas capacidades para reconocer patrones que no se ajustan a una expresión regular, sino más bien a una gramática de contexto libre. Esto es especialmente útil en determinadas circunstancias, como p.ej. para reconocer los comentarios de final de línea, admitidos por algunos lenguajes de programación (los

[1] La expresión ε representa a la cadena vacía.

que suelen comenzar por // y se extienden hasta el final de la línea actual.

$: el patrón que le precede sólo se reconoce si está al final de la línea. Ej.: (a|b|cd)$. Que el lexema se encuentre al final de la línea quiere decir que viene seguido por un retorno de carro, o bien por EOF. El carácter que identifica el final de la línea no forma parte del lexema.

^: fuera de los corchetes indica que el patrón que le sucede sólo se reconoce si está al comienzo de la línea. Que el lexema se encuentre al principio de la línea quiere decir que viene precedido de un retorno de carro, o que se encuentra al principio del fichero. El retorno de carro no pasa a formar parte del lexema. Nótese como las dos premisas de Lex hacen que los patrones de sensibilidad al contexto deban ponerse en primer lugar en caso de que existan ambigüedades:

Programa 1	Programa 2
^"casa"	"casa"
"casa"	^"casa"

En ambos programas el lexema "casa" cuando se encuentra al comienzo de línea puede entrar por ambos patrones luego, al existir ambigüedad, Lex selecciona el primero. Por ello, nótese cómo en el primer programa se entra por el patrón adecuado mientras que en el segundo se entra por el patrón general con lo que nunca se hará uso del patrón ^"casa".

/: Reconoce el patrón que le precede si y sólo si es prefijo de una secuencia simple como la que le sucede. Ej.: ab/c; en este caso si a la entrada se tiene "abcd", se reconocería el lexema "ab" porque está sucedido de 'c'. Si la entrada fuera "abdc" el patrón no se aplicaría. Por otro lado, un patrón como ab/c+ es erróneo puesto que el patrón c+ no se considera simple.

2.4.3.4 Estado léxicos

Los estados léxicos vienen a ser como variables lógicas excluyentes (una y sólo una puede estar activa cada vez) que sirven para indicar que un patrón sólo puede aplicarse si el estado léxico que lleva asociado se encuentra activado. En los ejemplos que hemos visto hasta ahora hemos trabajado con el estado léxico por defecto que, también por defecto, se encuentra activo al comenzar el trabajo de **yylex()**. Por ser un estado léxico por defecto no hemos tenido que hacer ninguna referencia explícita a él.

Los distintos estados léxicos (también llamados condiciones *start*) se declaran en el área de definiciones Lex de la forma:

%START id_1, id_2, ...

Una acción asociada a un patrón puede activar un estado léxico ejecutando la macro **BEGIN**, que es suministrada por el programa generado por PCLex. Así:

BEGIN id_i;

activaría la condición *start* id_i, que ha debido ser previamente declarada. La activación de un estado léxico produce automáticamente la desactivación de todos los demás. Por

otro lado, el estado léxico por defecto puede activarse con:

```
BEGIN 0;
```

Para indicar que un patrón sólo es candidato a aplicarse en caso de que se encuentre activa la condición *start* id_i, se le antepone la cadena "<id_i>". Si un patrón no tiene asociada condición *start* explícita, se asume que tiene asociado el estado léxico por defecto.

El siguiente ejemplo muestra cómo visualizar todos los nombres de los procedimientos y funciones de un programa Modula-2:

```
%START PROC
%%
"PROCEDURE" {BEGIN PROC;}
<PROC> [a-zA-Z][a-zA-Z0-9]* {     printf ("%s\n", yytext);
                                  BEGIN 0 ; }
```

2.4.3.5 Área de definiciones y área de funciones

El área de definiciones de un programa Lex tiene tres utilidades fundamentales:

a) Definir los estados léxicos.

b) Asignar un nombre a los patrones más frecuentes.

c) Poner código C que será global a todo el programa.

Ya hemos visto cómo definir estados léxicos, por lo que pasaremos a estudiar las otras dos posibilidades.

En el área de definiciones podemos crear expresiones regulares auxiliares de uso frecuente y asignarles un nombre. Posteriormente, estos patrones pueden ser referenciados en el área de reglas sin más que especificar su nombre entre llaves: {}. Por ejemplo, los siguientes dos programas son equivalentes, pero el primero es más claro y sus reglas más concisas:

Programa 1

```
D       [0-9]
L       [a-zA-Z]
%%
{D}+
{L}({L}|{D})*
```

Programa 2

```
%%
[0-9]+
[a-zA-Z][a-zA-Z0-9]*
```

Recordemos que en Lex, todo debe comenzar en la primera columna. Si algo no comienza en dicha columna, PCLex lo pasa directamente al programa C generado, sin procesarlo. De esta forma es posible crear definiciones de variables, etc. Sin embargo, esto no se recomienda ya que Lex posee un bloque específico para esto en el que se pueden poner incluso directivas del procesador, que deben comenzar obligatoriamente en la primera columna.

Así, el área de definiciones también permite colocar código C puro que se trasladará tal cual al comienzo del programa en C generado por PCLex. Para ello se

usan los delimitadores "%{" y "%}". Ej.:

```
%{
#include <stdlib.h>
typedef struct _Nodo{
        int valor;
        Nodo * siguiente;
} Nodo;
Nodo * tablaDeSimbolos;
%}
```

Es normal poner en esta zona las declaraciones de variables globales utilizadas en las acciones del área de reglas y un **#include** del fichero que implementa la tabla de símbolos.

Probablemente, el lector se estará preguntando qué contiene la función **main()** del programa C que genera PCLex. La verdad es que PCLex no genera **main()** alguno, sino que éste debe ser especificado por el programador. Por regla general, cuando se pretende ejecutar aisladamente un analizador léxico (sin un sintáctico asociado), lo normal es que las acciones asociadas a cada regla no contengan ningún **return** y que se incluya un **main()** que únicamente invoque a yylex():

```
void main(){
        yylex();
};
```

Para especificar el **main()** y cuantas funciones adicionales se estimen oportunas, se dispone del área de funciones. Dicha área es íntegramente copiada por PCLex en el fichero C generado. Es por ello que, a efectos prácticos, escribir código entre "%{" y "%}" en el área de definiciones es equivalente a escribirlo en el área de funciones.

El siguiente ejemplo informa de cuántas veces aparece el literal "resultado" en la cadena de entrada.

```
%{
        int cont=0;
%}
%%
"resultado"     {cont ++;}
. | \n    {;}
%%
void main(){
        yylex();
        printf("resultado aparece %d veces", cont);
}
```

2.4.3.6 Funciones y variables suministradas por PCLex

Como ya sabemos, el núcleo básico del programa en C generado por PCLex

es la función **yylex()**, que se encarga de buscar un lexema y ejecutar su acción asociada. Este proceso lo realiza iterativamente hasta que en una de las acciones se encuentre un **return** o se acabe la entrada. Además, PCLex suministra otras funciones macros y variables de apoyo; a continuación se muestra la lista de las más interesantes por orden de importancia:

yylex(): implementa el analizador lexicográfico.

yytext: contiene el lexema actual

yyleng: número de caracteres del lexema actual.

yylval: es una variable global que permite la comunicación con el sintáctico. Realmente no la define PCLex sino la herramienta PCYacc que se estudiará en capítulos posteriores.

yyin: es de tipo *FILE, y apunta al fichero de entrada que se lee. Inicialmente apunta a **stdin**, por lo que el fichero de entrada coincide con la entrada estándar.

yyout: de tipo *FILE, apunta al fichero de salida (inicialmente **stdout**).

yyerror() : es una función que se encarga de emitir y controlar errores (saca mensajes de error por pantalla). Realmente es definida por PCYacc como se verá más adelante.

yylineno: variable de tipo entero que, curiosamente, debe ser creada, inicializada y actualizada por el programador. Sirve para mantener la cuenta del número de línea que se está procesando. Normalmente se inicializa a 1 y se incrementa cada vez que se encuentra un retorno de carro.

yywrap(): el algoritmo de **yylex()** llama automáticamente a esta macro cada vez que se encuentra con un EOF. La utilidad de esta macro reside en que puede ser redefinida (para ello hay que borrarla previamente con la directiva **#undef**). Esta macro debe devolver *false* (que en C se representa por el entero 0) en caso de que la cadena de entrada no haya finalizado realmente, lo que puede suceder en dos tipos de situaciones: a) se está procesando un fichero binario que puede contener por en medio el carácter EOF como uno más, en cuyo caso el verdadero final del fichero hay que descubrirlo contrastando la longitud del fichero con la longitud de la cadena consumida; y b) la cadena a reconocer se encuentra particionada en varios ficheros, en cuyo caso cuando se llega al final de uno de ellos, hay que cargar en **yyin** el siguiente y continuar el procesamiento. **yywrap()** debe devolver *true* (valor entero 1) en caso de que el EOF encontrado identifique realmente el final de la cadena a procesar.

yyless(int n): deja en **yytext** los **n** primeros caracteres del lexema actual. El resto los devuelve a la entrada, por lo que podemos decir que son des-consumidos. **yyleng** también se modifica convenientemente. Por ejemplo, el patrón `abc*/\n`

es equivalente a:

```
abc*\n   { yyless(yyleng-1); }
```

input(): consume el siguiente carácter de la entrada y lo añade al lexema actual. P,ej., el programa:

```
%%
abc      { printf ("%s", yytext);
           input( );
           printf("%s", yytext);}
```

ante la entrada "abcde" entraría por este patrón: el lexema antes del **input()** (en el primer **printf**) es "abc", y después del **input** (en el segundo **printf**) es "abcd".

output(char c): emite el carácter **c** por la salida estándar. PCLex para DOS no soporta esta función, pero sí el Lex para Unix.

unput(char c): des-consume el carácter **c** y lo coloca al comienzo de la entrada. P.ej., supongamos que el programa:

```
%%
abc      { printf ("%s",yytext); unput('t'); }
tal      { printf ("%s",yytext); }
```

recibe la entrada "abcal". Los tres primeros caracteres del lexema coinciden con el primer patrón. Después queda en la entrada la cadena "al", pero como se ha hecho un **unput('t')**, lo que hay realmente a la entrada es "tal", que coincide con el segundo patrón.

ECHO: macro que copia la entrada en la salida (es la acción que se aplica por defecto a los lexemas que no encajan por ningún patrón explícito).

yymore(): permite pasar a reconocer el siguiente lexema, pero sin borrar el contenido actual de **yytext**, por lo que el nuevo lexema leído se concatena al ya existente. PCLex para DOS no soporta esta función, pero sí el Lex para Unix.

La utilidad principal de **yymore()** reside en que facilita el reconocer los literales entrecomillados. Supongamos que se desea construir un patrón para reconocer literales entrecomillados. Una primera idea podría ser el patrón:

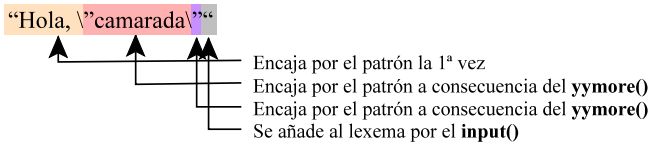
```
\"[^\"]*\"          /* Lexemas que empiecen por comillas,
                       cualquier cosa, y termine en comillas. */
```

que reconocería cadenas como "Hola", "adiós", e incluso "Ho;*". Sin embargo una cadena como "Hola, \"camarada\"" que, a su vez contiene comillas dentro, dará problemas porque la primera comilla de "camarada", es considerada como unas comillas de cierre. La solución a este problema pasa por el siguiente bloque de código Lex:

```
\"[^\"]*/\"          { if (yytext[yyleng-1]=='\\')
                            yymore();
```

```
                else
                    input();}
```

Este patrón reconoce cadenas entrecomilladas (excepto las últimas comillas), de forma que las cadenas que tienen dentro comillas se reconocen por partes. Siguiendo con el ejemplo de antes, "Hola, \"camarada\""" se reconocería como:



Este patrón fallaría si los caracteres anteriores a las comillas de cierre fuesen precisamente '\\'. Para solucionarlo habría que hacer uso de **unput** y de una variable global que actuase como *flag*. Cualquier sentencia que haya detrás de **yymore()** nunca se ejecutará, ya que actúa como una especie de **goto**.

REJECT: rechaza el lexema actual, lo devuelve a la entrada y busca otro patrón. **REJECT** le dice a **yylex()** que el lexema encontrado no corresponde realmente a la expresión regular en curso, y que busque la siguiente expresión regular a que corresponda. La macro **REJECT** debe ser lo último que una acción ejecute puesto que si hay algo detrás, no se ejecutará. P.ej. Para buscar cuántas veces aparecen las palabras "teclado" y "lado", se puede construir el siguiente programa Lex:

```
%{
        int t=0, l=0;
%}
%%
teclado         {t ++; REJECT;}
lado            {l ++;}
```

Ante la entrada "teclado" este programa se comporta de la siguiente forma:

1.- Entra por el patrón que lee el lexema más largo que sería "teclado" y ejecuta la acción asociada: se incrementa la variable **t** y se rechaza el lexema actual, por lo que el texto entero se devuelve a la entrada.

2.- Saca por pantalla "tec" que es la acción por defecto cuando se encuentran caracteres que no encajan con ningún patrón.

3.- El lexema "lado"' coincide con el segundo patrón y ejecuta la acción asociada: se incrementa la variable **l**.

Para finalizar, resulta evidente que el programador no debe declarar ninguna función o variable cuyo nombre coincida con las que acabamos de estudiar. De hecho PCLex también genera una serie de variables auxiliares cuyo nombre consta de un solo carácter, por lo que tampoco es bueno declarar variables o funciones con nombres de una sola letra ni que empiecen por **yy**.

2.5 El generador de analizadores lexicográficos JFlex

JFlex es un generador de analizadores lexicográficos desarrollado por Gerwin Klein como extensión a la herramienta JLex desarrollada en la Universidad de Princeton. JFlex está desarrollado en Java y genera código Java.

Los programas escritos para JFlex tienen un formato parecido a los escritos en PCLex; de hecho todos los patrones regulares admisibles en Lex también son admitidos por JFlex, por lo que en este apartado nos centraremos tan sólo en las diferencias y extensiones, tanto de patrones como del esqueleto que debe poseer el fichero de entrada a JFlex.

La instalación y ejecución de JFlex es trivial. Una vez descomprimido el fichero **jflex-1.3.5.zip**, dispondremos del fichero JFlex.jar que tan sólo es necesario en tiempo de meta-compilación, siendo el analizador generado totalmente independiente. La clase Main del paquete JFlex es la que se encarga de metacompilar nuestro programa .jflex de entrada; de esta manera, una invocación típica es de la forma:

java JFlex.Main fichero.jflex

lo que generará un fichero Yylex.java que implementa al analizador lexicográfico. La figura 2.5 ilustra el proceso a seguir, y cómo el nombre por defecto de la clase generada es **Yylex**.

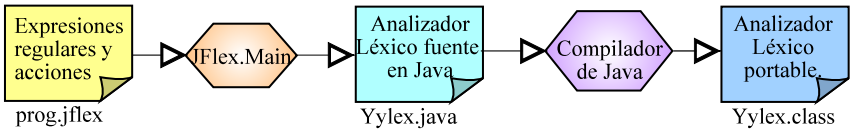


Figura 2.5. Obtención de un programa portable en Java a partir de una especificación JFlex

2.5.1 Ejemplo preliminar

Resulta interesante comenzar con un ejemplo preliminar sobre el que poder discutir. El ejemplo (almacenado en un fichero de texto llamado **prueba.jflex**) es el siguiente:

```
/* Ejemplo de JFlex */
import java.io.*;
%%
// %class Lexer
%int
%unicode
// %cup
%line
%column
%{
    public static void main(String[] args){
        Yylex analizadorLexico =
```

```
                new Yylex(new InputStreamReader(System.in));
            try{
                analizadorLexico.yylex();
            }catch(IOException x){
                System.out.println("Error en la línea "+
                                    analizadorLexico.yyline+
                                    " columna "+
                                    analizadorLexico.yycolumn);
            }
        }
%}
terminadorLinea = \r|\n|\r\n
espacioBlanco = {terminadorLinea} | [ \t\f]
%state STRING
%%
<YYINITIAL>"IF"  { System.out.println("Encontrado IF"); }
<YYINITIAL>{
    "="             { System.out.println("Encontrado ="); }
    "!="        { System.out.println("Encontrado !="); }
    "THEN"      { System.out.println("Encontrado THEN"); }
    {espacioBlanco}     { /* Ignorar */; }
    \"              { yybegin(STRING); }
    [:jletter:][:jletterdigit:]* { System.out.println("Encontrado un ID"); }
}
<STRING>{
    [^\n\"\\]   { System.out.println("Estoy en una cadena"); }
    \n          { System.out.println("Una cadena no puede acabar en \\n");}
    \\(.|\n)    { System.out.println("Encontrado "+ yytext() + " en cadena");}
    \"          { yybegin(YYINITIAL); }
}
.|\n            { System.out.println("Encontrado cualquier carácter"); }
```

Como puede observarse en este ejemplo, el texto se encuentra dividido en tres secciones separadas por **%%**, al igual que en Lex. Sin embargo, el contenido de cada sección o área difiere con respecto a aquél, siendo la estructura general de la forma:

```
Área de código, importaciones y paquete
%%
Área de opciones y declaraciones
%%
Área de reglas
```

La primera de estas áreas se encuentra destinada a la importación de los paquetes que se vayan a utilizar en las acciones regulares situadas al lado de cada patrón en la zona de reglas. Aquí también puede indicarse una cláusula **package** para los ficheros **.java** generados por el meta-analizador.

En general, cualquier cosa que se escriba en este área se trasladará tal cual al fichero .**java** generado por JFlex., por lo que también sirve para escribir clases

completas e interfaces.

A continuación se describe en profundidad las siguientes dos áreas.

2.5.2 Área de opciones y declaraciones

Este área permite indicar a JFlex una serie de opciones para adaptar el fichero **.java** resultante de la meta-compilación y que será el que implemente nuestro analizador lexicográfico en Java. También permite asociar un identificador de usuario a los patrones más utilizados en el área de reglas.

2.5.2.1 Opciones

Las opciones más interesantes se pueden clasificar en opciones de clase, de la función de análisis, de fin de fichero, de juego de caracteres y de contadores. Todas ellas empiezan por el carácter **%** y no pueden estar precedidas por nada en la línea en que aparecen.

2.5.2.1.1 Opciones de clase

Las opciones de clase más útiles son:

%class nombreClase. Por defecto, la clase que genera JFlex y que implementa el analizador lexicográfico se llama **Yylex** y se escribe, por tanto, en un fichero **Yylex.java**. Utilizando esta opción puede cambiarse el nombre de dicha clase.

%implements interface1, interface2, etc. Genera una clase (por defecto **Yylex**) que implementa las interfaces indicadas. El programador deberá introducir los métodos necesarios para hacer efectiva dicha implementación.

%extends nombreClase. Genera una clase que hereda de la clase indicada.

%public. Genera una clase pública. La clase generada por defecto no posee modificar de ámbito de visibilidad.

%final. Genera una clase final, de la que nadie podrá heredar.

%abstract. Genera una clase abstracta de la que no se pueden crear objetos.

%{ bloqueJava %}.El programador también puede incluir sus propias declaraciones y métodos en el interior de la clase **Yylex** escribiendo éstas entre los símbolos **%{** y **%}**.

%init{ códigoJava %init}. Es posible incluir código en el constructor generado automáticamente por JFlex para la clase **Yylex** mediante el uso de esta opción.

2.5.2.1.2 Opciones de la función de análisis

Estas opciones permiten modificar el método o función encargada de realizar el análisis lexicográfico en sí. Las opciones más útiles son:

%function nombreFuncionAnalizadora. Por defecto, la función que arranca el análisis se llama **yylex()** y devuelve un valor de tipo **Yytoken**. Esta opción permite cambiar de nombre a la función **yylex()**.

%int. Hace que la función **yylex()** devuelva valores de tipo **int** en lugar de **Yytoken**.

%intwrap. Hace que la función **yylex()** devuelva valores de tipo **Integer** en lugar de **Yytoken**.

%type nombreTipo. Hace que la función **yylex()** devuelva valores del tipo especificado (ya sea primitivo o no) en lugar de **Yytoken**.

En cualquier caso, JFlex no crea la clase **Yytoken**, sino que debe ser suministrada de forma externa, o bien especificada en el área de código.

2.5.2.1.3 Opciones de fin de fichero

Por defecto, cuando **yylex()** se encuentra el carácter fin de fichero, retorna un valor **null**. En caso de que se haya especificado **%int** en las opciones anteriores, se retornará el valor **YYEOF** definido como **public static final** en la clase **Yylex**. Para cambiar este comportamiento, las opciones de fin de fichero más útiles son:

%eofclose. Hace que **yylex()** cierre el canal de lectura en cuanto se encuentre el EOF.

%eofval{ códigoJava %eofval}. Es posible ejecutar un bloque de código cuando **yylex()** se encuentre el EOF. Para ello se utilizará esta opción.

2.5.2.1.4 Opciones de juego de caracteres

Estas opciones permiten especificar el juego de caracteres en el que estará codificada la entrada al analizador lexicográfico generado por JFlex. Las posibilidades son:

%7bit. Es la opción por defecto, y asume que cada carácter de la entrada está formado por un único byte cuyo bit más significativo es 0, lo que da 128 caracteres.

%8bit. Asume que cada carácter está formado por un byte completo, lo que da 256 caracteres.

%unicode. Asume que la entrada estará formada por caracteres Unicode. Esto no quiere decir que se tomen dos *bytes* de la entrada por cada carácter sino que la recuperación de caracteres se deja recaer en la plataforma sobre la que se ejecuta **yylex().**

%ignorecase. Hace que **yylex()** ignore entre mayúsculas y minúsculas mediante el uso de los métodos **toUpperCase** y **toLowerCase** de la clase **Character**.

Resulta evidente que las tres primeras opciones son mutuamente excluyentes.

2.5.2.1.5 Opciones de contadores

Estas opciones hacen que el analizador lexicográfico almacene en contadores el número de caracteres, líneas o columnas en la que comienza el lexema actual. La nomenclatura es:

%char. Almacena en la variable **yychar** el número de caracteres que hay entre el comienzo del canal de entrada y el comienzo del lexema actual.

%line. Almacena en la variable **yyline** el número de línea en que comienza el lexema actual.

%column. almacena en la variable **yycolumn** el número de columna en que comienza el lexema actual.

Todos los contadores se inician en 0.

2.5.2.2 Declaraciones.

Además de opciones, el programador puede indicar declaraciones de dos tipos en el área que nos ocupa, a saber, declaraciones de estados léxicos y declaraciones de reglas.

2.5.2.2.1 Declaraciones de estados léxicos.

Los estado léxicos se declaran mediante la opción:

```
%state estado1, estado2, etc.
```

y pueden usarse en el área de reglas de la misma manera que en Lex. Además, si hay muchos patrones en el área de reglas que comparten el mismo estado léxico, es posible indicar éste una sola vez y agrupar a continuación todas estas reglas entre llaves, como puede observarse en el ejemplo preliminar del punto 2.5.1. El estado inicial viene dado por la constante **YYINITIAL** y, a diferencia de Lex, puede utilizarse como cualquier otro estado léxico definido por el usuario.

Una acción léxica puede cambiar de estado léxico invocando a la función **yybegin(estadoLéxico)** generada por JFlex.

2.5.2.2.2 Declaraciones de reglas.

En caso de que un patrón se utilice repetidas veces o cuando su complejidad es elevada, es posible asignarle un nombre y utilizarlo posteriormente en cualquier otra regla encerrándolo entre llaves, de manera análoga a como se estudió en Lex. La sintaxis es:

```
nombre = patron
```

2.5.3 Área de reglas.

El área de reglas tiene la misma estructura que en Lex, con la única diferencia de que es posible agrupar las reglas a aplicar en un mismo estado léxico. Como ejemplo, las reglas:

```
<YYINITIAL>"="          { System.out.println("Encontrado ="); }
<YYINITIAL>"!="         { System.out.println("Encontrado !="); }
```

también pueden escribirse como:

```
<YYINITIAL>{
        "="             { System.out.println("Encontrado ="); }
        "!="            { System.out.println("Encontrado !="); }
}
```

Las diferencias importantes de este área con respecto a Lex se concentran en que JFlex incluye algunas características adicionales para la especificación de patrones, como son:

\u**nºhexadecimal**: representa el carácter cuyo valor Unicode es **nºhexadecimal**. Ej.: \u042D representa al carácter "Э".

!: encaja con cualquier lexema excepto con los que entran por el patrón que le sucede. Ej.: !(ab) encaja con cualquier cosa excepto el lexema "ab".

~: encaja con un lexema que empieza con cualquier cosa y acaba con la primera aparición de un texto que encaja con el patrón que le sigue. P.ej., la manera más fácil de reconocer un comentario al estilo de Java (no anidado) viene dada por el patrón: "/*"~"*/"

Por otro lado JFlex incluye algunas clases de caracteres predefinidas. Estas clases van englobadas entre los símbolo [: y :] y cada una encaja con el conjunto de caracteres que satisfacen un determinado predicado. Por ejemplo, el patrón [:digit:] encaja con aquellos caracteres que satisfacen la función lógica **isDigit()** de la clase **Character**. La lista de patrones es:

Patrón	Predicado asociado
[:jletter:]	isJavaIdentifierStart()
[:jletterdigit:]	isJavaIdentifierPart()
[:letter:]	isLetter()
[:digit:]	isDigit()
[:uppercase:]	isUpperCase()
[:lowercase:]	isLowerCase()

Finalmente, otras dos diferencias importantes con respecto a Lex son:

- Obligatoriamente toda regla debe tener una acción léxica asociada, aunque sea vacía.
- Si un lexema no encaja por ningún patrón se produce un error léxico.
- El patrón \n engloba sólo al carácter de código ASCII 10, pero no al de código ASCII 13, que viene representado por el patrón \r.

2.5.4 Funciones y variables de la clase Yylex

Ya hemos visto dos de las funciones más importantes generadas por JFlex y que pueden ser utilizadas por el programador en el interior de las acciones léxicas. Se trata de funciones y variables miembro, por lo que, si son utilizadas fuera de las acciones léxicas deberá indicarse el objeto contextual al que se aplican (p.ej. **miAnalizador.yylex()**). Aquí las recordamos y las extendemos:

Yylex(Reader r): es el constructor del analizador léxico. Toma como parámetro el canal de entrada del cual se leerán los caracteres.

Yytoken yylex(): función principal que implementa el analizador léxico. Toma la entrada del parámetro especificado en la llamada al constructor de **Yylex**. Puede elevar una excepción **IOException**, por lo que se recomienda invocarla en el interior de una sentencia **try-catch**.

String yytext(): devuelve el lexema actual.

int yylength(): devuelve el número de caracteres del lexema actual.

void yyreset(Reader r): cierra el canal de entrada actual y redirige la entrada hacia el nuevo canal especificado como parámetro.

void yypushStream(Reader r): guarda el canal de entrada actual en una pila y continúa la lectura por el nuevo canal especificado. Cuando éste finalice continuará por el anterior, extrayéndolo de la pila. Esta función es de especial importancia para implementar la directiva **#include** de lenguajes como C.

void yypushback(int n): equivale a la función **yyless()** de Lex.

yyline, **yychar** e **yycolumn**: son las variables generadas por las opciones **%line**, **%char** y **%column** respectivamente, y comentadas en el apartado 2.5.2.1.5.

Capítulo 3

Análisis sintáctico

3.1 Visión general

Todo lenguaje de programación obedece a unas reglas que describen la estructura sintáctica de los programas bien formados que acepta. En Pascal, por ejemplo, un programa se compone de bloques; un bloque, de sentencias; una sentencia, de expresiones; una expresión, de componentes léxicos; y así sucesivamente hasta llegar a los caracteres básicos. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o utilizando notación BNF (*Backus-Naur Form*).

Las gramáticas formales ofrecen ventajas significativas a los diseñadores de lenguajes y a los desarrolladores de compiladores:

- Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.
- A partir de una gramática se puede generar automáticamente un analizador sintáctico.
- El proceso de generación automática anterior puede llevar a descubrir ambigüedades.
- Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
- Es más fácil ampliar/modificar el lenguaje si está descrito con una gramática.

La mayor parte del presente tema está dedicada a los métodos de análisis sintáctico de uso típico en compiladores. Comenzaremos con los conceptos básicos y las técnicas adecuadas para la aplicación manual. A continuación trataremos la gestión y recuperación de errores sintácticos. Estudiaremos los métodos formales de análisis mediante autómatas con pila y, por último, profundizaremos en el funcionamiento de metacompiladores que generan automáticamente el analizador sintáctico de un lenguaje.

3.2 Concepto de analizador sintáctico

Es la fase del analizador que se encarga de chequear la secuencia de *tokens* que representa al texto de entrada, en base a una gramática dada. En caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce en base a una representación computacional. Este árbol es el punto de partida de la fase posterior de la etapa de análisis: el analizador semántico.

Pero esto es la teoría; en la práctica, el analizador sintáctico dirige el proceso de compilación, de manera que el resto de fases evolucionan a medida que el sintáctico va reconociendo la secuencia de entrada por lo que, a menudo, el árbol ni siquiera se genera realmente.

En la práctica, el analizador sintáctico también:

- Incorpora acciones semánticas en las que colocar el resto de fases del compilador (excepto el analizador léxico): desde el análisis semántico hasta la generación de código.
- Informa de la naturaleza de los errores sintácticos que encuentra e intenta recuperarse de ellos para continuar la compilación.
- Controla el flujo de *tokens* reconocidos por parte del analizador léxico.

En definitiva, realiza casi todas las operaciones de la compilación, dando lugar a un método de trabajo denominado **compilación dirigida por sintaxis**.

3.3 Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificarían mucho. Las primeras versiones de los programas suelen ser incorrectas, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.
- De corrección, cuando el programa no hace lo que el programador realmente deseaba.

Resulta evidente que los errores de corrección no pueden ser detectados por un compilador, ya que en ellos interviene el concepto abstracto que el programador tiene sobre el programa que construye, lo cual es desconocido, y probablemente incognoscible, por el compilador. Por otro lado, la detección de errores lógicos implica un esfuerzo computacional muy grande en tanto que el compilador debe ser capaz de averiguar los distintos flujos que puede seguir un programa en ejecución lo cual, en muchos casos, no sólo es costoso, sino también imposible. Por todo esto, los compiladores actuales se centran en el reconocimiento de los tres primeros tipos de errores. En este tema hablaremos de los errores de sintaxis, que son los que pueden impedir la correcta construcción de un árbol sintáctico.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, no cancele definitivamente la compilación, sino que se recupere y siga buscando errores. Recuperar un error no quiere decir corregirlo, sino ser capaz de seguir construyendo el árbol sintáctico a pesar de los errores encontrados. En vista de esto, el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Debe informar mediante los correspondientes mensajes del tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.
- Distinguir entre errores y advertencias. Las advertencias se suelen utilizar para informar sobre sentencias válidas pero que, por ser poco frecuentes, pueden constituir una fuente de errores lógicos.
- No ralentizar significativamente la compilación.

Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir, con lo que se consigue simplificar su estructura. Además, si el propio compilador está preparado para admitir incluso los errores más frecuentes, entonces se puede mejorar la respuesta ante esos errores incluso corrigiéndolos.

A continuación se estudiarán varias estrategias para gestionar los errores una vez detectados

3.3.1 Ignorar el problema

Esta estrategia (denominada *panic mode* en inglés) consiste en ignorar el resto de la entrada hasta llegar a una condición de seguridad. Una condición tal se produce cuando nos encontramos un *token* especial (por ejemplo un ';' o un 'END'). A partir de este punto se sigue analizando normalmente. Los *tokens* encontrados desde la detección del error hasta la condición del error son desechados, así como la secuencia de *tokens* previa al error que se estime oportuna (normalmente hasta la anterior condición de seguridad).

Por ejemplo, supongamos la siguiente secuencia de sentencias de asignación en C o Java:

```
aux = a[i]
a[i] = a[j];
a[j] = aux;
```

La secuencia que se encuentra el analizador sintáctico aparece en la figura 3.1; claramente, falta el punto y coma de la primera sentencia y, dado que no tiene sentido una sentencia como:

```
aux = a[i] a[i] = a[j];
```

pues el analizador sintáctico advierte un error, y aprovecha que el ';' se usa como terminador de sentencias para ignorar el resto de la entrada hasta el siguiente ';'. Algunas consideraciones adicionales sobre esta estrategia de gestión de errores pueden estudiarse en los puntos 4.3.4 y 8.4.1.

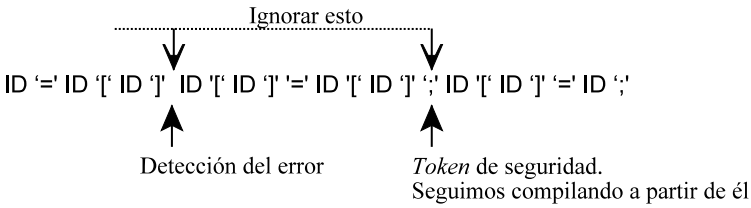


Figura 3.1. Recuperación de error sintáctico ignorando la secuencia errónea

3.3.2 Recuperación a nivel de frase

Intenta corregir el error una vez descubierto. P.ej., en el caso propuesto en el punto anterior, podría haber sido lo suficientemente inteligente como para insertar el *token* ';' . Hay que tener cuidado con este método, pues puede dar lugar a recuperaciones infinitas, esto es, situaciones en las que el intento de corrección no es el acertado, sino que introduce un nuevo error que, a su vez, se intenta corregir de la misma manera equivocada y así sucesivamente.

3.3.3 Reglas de producción adicionales

Este mecanismo añade a la gramática formal que describe el lenguaje reglas de producción para reconocer los errores más comunes. Siguiendo con el caso del punto 3.3.1 , se podría haber puesto algo como:

sent_errónea → sent_sin_acabar sent_acabada
sent_acabada → sentencia ';'
sent_sin_acabar → sentencia

lo cual nos da mayor control, e incluso permite recuperar y corregir el problema y emitir un mensaje de advertencia en lugar de uno de error.

3.3.4 Corrección Global

Este método trata por todos los medios de obtener un árbol sintáctico para una secuencia de *tokens*. Si hay algún error y la secuencia no se puede reconocer, entonces este método infiere una secuencia de *tokens* sintácticamente correcta lo más parecida a la original y genera el árbol para dicha secuencia. Es decir, el analizador sintáctico le pide toda la secuencia de *tokens* al léxico, y lo que hace es devolver lo más parecido a la cadena de entrada pero sin errores, así como el árbol que lo reconoce.

3.4 Gramática utilizada por un analizador sintáctico

Centraremos nuestros esfuerzos en el estudio de análisis sintácticos para lenguajes basados en gramáticas formales, ya que de otra forma se hace muy difícil la comprensión del compilador. La formalización del lenguaje que acepta un compilador sistematiza su construcción y permite corregir, quizás más fácilmente, errores de muy difícil localización, como es la ambigüedad en el reconocimiento de ciertas sentencias.

La gramática que acepta el analizador sintáctico es una gramática de contexto libre, puesto que no es fácil comprender gramáticas más complejas ni construir automáticamente autómatas reducidos que reconozcan las sentencias que aceptan. Recuérdese que una gramática G queda definida por una tupla de cuatro elementos (N, T, P, S), donde:

N = No terminales.
T = Terminales.
P = Reglas de Producción.
S = Axioma Inicial.

Por ejemplo, una gramática no ambigua que reconoce las operaciones aritméticas podría ser la del cuadro 3.1. En ésta se tiene que N = {E, T, F} ya que E, T y F aparecen a la izquierda de alguna regla; T = {id, num, (,) ,+ ,*}; P son las siete reglas de producción, y S = E, pues por defecto el axioma inicial es el antecedente de la primera regla de producción.

①	E	→	E + T
②		|	T
③	T	→	T * F
④		|	F
⑤	F	→	id
⑥		|	num
⑦		|	(E)

Cuadro 3.1 Gramática no ambigua que reconoce expresiones aritméticas

3.4.1 Derivaciones

Una regla de producción puede considerarse como equivalente a una regla de reescritura, donde el no terminal de la izquierda es sustituido por la pseudocadena del lado derecho de la producción. Podemos considerar que una pseudocadena es cualquier secuencia de terminales y/o no terminales. Formalmente, se dice que una pseudocadena α deriva en una pseudocadena β, y se denota por $\alpha \Rightarrow \beta$, cuando:

$$\alpha \Rightarrow \beta \text{ con } \alpha, \beta \in (N \cup T)^*, \text{ y siendo:}$$
$$\left.\begin{array}{l}\alpha \equiv \sigma\ A\ \delta \\ \beta \equiv \sigma\ \tau\ \delta\end{array}\right\} \text{ donde } A \in N \wedge \sigma, \tau, \delta \in (N \cup T)^* \wedge A \rightarrow \tau \in P$$

Nótese que dentro de un ‘α’ puede haber varios no terminales ‘A’ que pueden ser reescritos, lo que da lugar a varias derivaciones posibles. Esto hace que aparezcan los siguientes conceptos:

- Derivación por la izquierda: es aquella en la que la reescritura se realiza sobre el no terminal más a la izquierda de la pseudocadena de partida.
- Derivación por la derecha: es aquella en la que la reescritura se realiza sobre el no terminal más a la derecha de la pseudocadena de partida.

Por ejemplo, partiendo de la gramática del cuadro 3.2, vamos a realizar todas las derivaciones a derecha e izquierda que podamos, a partir del axioma inicial, y teniendo como objetivo construir la cadena de *tokens*: (id_1 * id_2 + id_3). En cada pseudocadena aparece apuntado por una flechita el no terminal que se escoge para realizar la siguiente derivación.

① E	→	E + E
②	|	E * E
③	|	num
④	|	id
⑤	|	(E)

Cuadro 3.2 Gramática ambigua que reconoce expresiones aritméticas

Las derivaciones a izquierda quedarían:

$$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id_1 * E + E \Rightarrow id_1 * id_2 + E \Rightarrow id_1 * id_2 + id_3$$
$$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$$

Mientras que las de a derecha darían lugar a:

$$E \Rightarrow E + E \Rightarrow E + id_3 \Rightarrow E * E + id_3 \Rightarrow E * id_2 + id_3 \Rightarrow id_1 * id_2 + id_3$$
$$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$$

Las pseudocadenas que se originan tras una secuencia cualquiera de derivaciones a partir del axioma inicial reciben el nombre de **formas sentenciales**.

3.4.2 Árbol sintáctico de una sentencia de un lenguaje

Al igual que podemos analizar una sentencia en español y obtener su árbol sintáctico, también es posible hacerlo con una sentencia de un lenguaje de programación. Básicamente un árbol sintáctico se corresponde con una sentencia, obedece a una gramática, y constituye una representación que se utiliza para describir el proceso de derivación de dicha sentencia. La raíz del árbol es el axioma inicial y, según nos convenga, lo dibujaremos en la cima o en el fondo del árbol.

Como nodos internos del árbol, se sitúan los elementos no terminales de las reglas de producción que vayamos aplicando, y cada uno de ellos poseerá tantos hijos como símbolos existan en la parte derecha de la regla aplicada.

Veamos un ejemplo utilizando la gramática del cuadro 3.1; supongamos que hay que reconocer la cadena (a + b) * a + b que el analizador léxico nos ha suministrado como (id_a + id_b) * id_a + id_b y vamos a construir el árbol sintáctico, que sería el de la figura 3.2.

Nótese que una secuencia de derivaciones con origen en el axioma inicial y destino en la cadena a reconocer constituye una representación del árbol sintáctico. A cada paso de derivación se añade un nuevo trozo del árbol, de forma que al comienzo del proceso se parte de un árbol que sólo tiene la raíz (el axioma inicial), al igual que

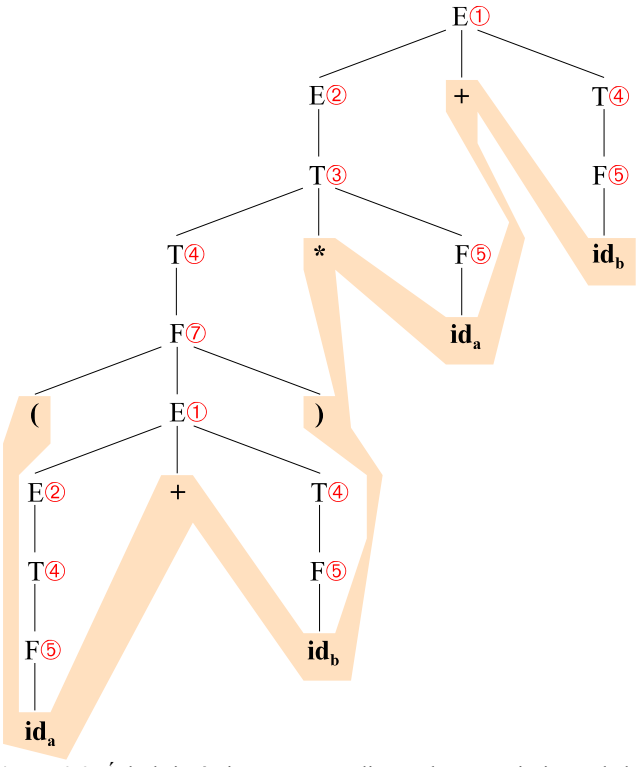


Figura 3.2. Árbol sintáctico correspondiente al reconocimiento de la secuencia (id_a + id_b) * id_a + id_b, que se halla marcada de color.

las derivaciones que hemos visto parten también del axioma inicial. De esta forma, en cada derivación se selecciona un nodo hoja del árbol que se está formando y se le añaden sus hijos. En cada paso, las hojas del árbol que se va construyendo constituyen la forma sentencial por la que discurre la derivación. Por otro lado, la secuencia de derivaciones no sólo representa a un árbol sintáctico, sino que también da información de en qué orden se han ido aplicando las reglas de producción.

La construcción de este árbol conlleva la aplicación inteligente de las reglas de producción. Si la sentencia a reconocer es incorrecta, esto es, no pertenece al lenguaje generado por la gramática, será imposible obtener su árbol sintáctico. En otras ocasiones una sentencia admite más de un árbol sintáctico. Cuando esto ocurre se dice que la gramática es ambigua. Este tipo de situaciones hay que evitarlas, de suerte que, o bien se cambia la gramática, o bien se aplican reglas *desambiguantes* (que veremos posteriormente). Por ejemplo, la gramática del cuadro 3.2 es ambigua puesto que ante una suma múltiple no permite distinguir entre asociatividad a la derecha o a la izquierda, esto es, ante la sentencia "aux + cont + i" que se traduce por los *tokens*: id_{aux}

+ id_{cont} + id_i se pueden obtener los dos árboles distintos de la figura 3.3 (hemos puesto uno arriba y otro abajo, aunque ambos se formarían de la misma manera).

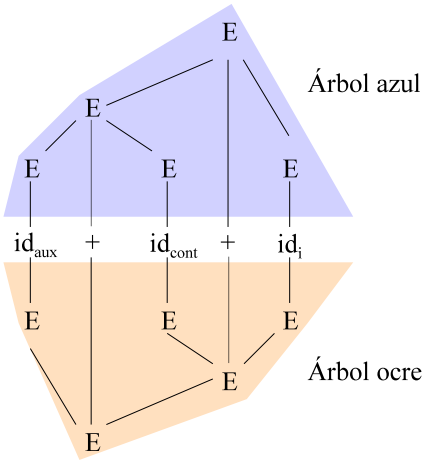


Figura 3.3. Ejemplo de reconocimiento de una sentencia mediante una gramática ambigua.

Los dos árboles obtenidos obedecen a dos secuencias de derivaciones distintas:

- Árbol azul:
  $E \Rightarrow E + E \Rightarrow E + id_i \Rightarrow E + E + id_i \Rightarrow E + id_{cont} + id_i \Rightarrow id_{aux} + id_{cont} + id_i$
  ↑ ↑ ↑ ↑ ↑
- Árbol ocre:
  $E \Rightarrow E + E \Rightarrow id_{aux} + E \Rightarrow id_{aux} + E + E \Rightarrow id_{aux} + id_{cont} + E \Rightarrow id_{aux} + id_{cont} + id_i$
  ↑ ↑ ↑ ↑ ↑

Como puede observarse, el quid de la cuestión está en la segunda derivación: en el árbol azul se hace una derivación a derecha, mientras que en el ocre es a izquierda; si los árboles se leen desde la cadena hacia el axioma inicial, el árbol azul da lugar a una asociatividad a la izquierda para la suma $((id_{aux} + id_{cont}) + id_i)$, mientras que el ocre produce asociatividad a derecha $(id_{aux} + (id_{cont} + id_i))$. Este hecho puede dar una idea de que la forma en que se construye el árbol sintáctico también conlleva aspectos semánticos.

3.5 Tipos de análisis sintáctico

Según la aproximación que se tome para construir el árbol sintáctico se desprenden dos tipos o clases de analizadores:

- Descendentes: parten del axioma inicial, y van efectuando derivaciones a izquierda hasta obtener la secuencia de derivaciones que reconoce a la sentencia. Pueden ser:
    - Con retroceso.
    - Con funciones recursivas.
    - De gramáticas LL(1).
- Ascendentes: Parten de la sentencia de entrada, y van aplicando derivaciones inversas (desde el consecuente hasta el antecedente), hasta llegar al axioma inicial. Pueden ser:
    - Con retroceso.
    - De gramáticas LR(1).

Estudiaremos a continuación con detenimiento cada uno de estos métodos.

3.5.1 Análisis descendente con retroceso

Al tratarse de un método descendente, se parte del axioma inicial y se aplican en secuencia todas las posibles reglas al no terminal más a la izquierda de la forma sentencial por la que se va trabajando.

Podemos decir que todas las sentencias de un lenguaje se encuentran como nodos de un árbol que representa todas las derivaciones posibles a partir de cualquier forma sentencial, y que tiene como raíz al axioma inicial. Éste es el **árbol universal** de una gramática, de manera que sus ramas son secuencias de derivaciones y, por tanto, representan árboles sintácticos. Básicamente, podemos decir que el método de análisis descendente con retroceso pretende buscar en el árbol universal a la sentencia a reconocer; cuando lo encuentre, el camino que lo separa de la raíz nos da el árbol sintáctico. Ahora bien, es posible que la sentencia sea errónea y que no se encuentre como hoja del árbol lo que, unido a que es muy probable que el árbol sea infinito, nos lleva a la necesidad de proponer un enunciado que nos indique cuándo se debe cancelar la búsqueda porque se da por infructuosa.

Para ver un ejemplo de esto vamos a partir de la gramática del cuadro 3.3 que es equivalente a la del cuadro 3.1, aunque la que se propone ahora no es recursiva por la izquierda. Y supongamos que se desea reconocer la secuencia:

①	E	→	T + E
②		|	T
③	T	→	F * T
④		|	F
⑤	F	→	id
⑥		|	num
⑦		|	(E)

Cuadro 3.3 Gramática no ambigua ni recursiva por la izquierda que reconoce expresiones aritméticas

(id + num) * id + num

Mediante el árbol de la figura 3.4 se pueden derivar todas las posibles sentencias reconocibles por la gramática propuesta y el algoritmo que sigue el análisis descendente con retroceso consiste hacer una búsqueda en este árbol de la rama que culmine en la sentencia a reconocer mediante un recorrido primero en profundidad.

¿Cuando se desecha una rama que posee la forma sentencial τ? Pues, p.ej., cuando la secuencia de *tokens* a la izquierda del primer no terminal de τ no coincide

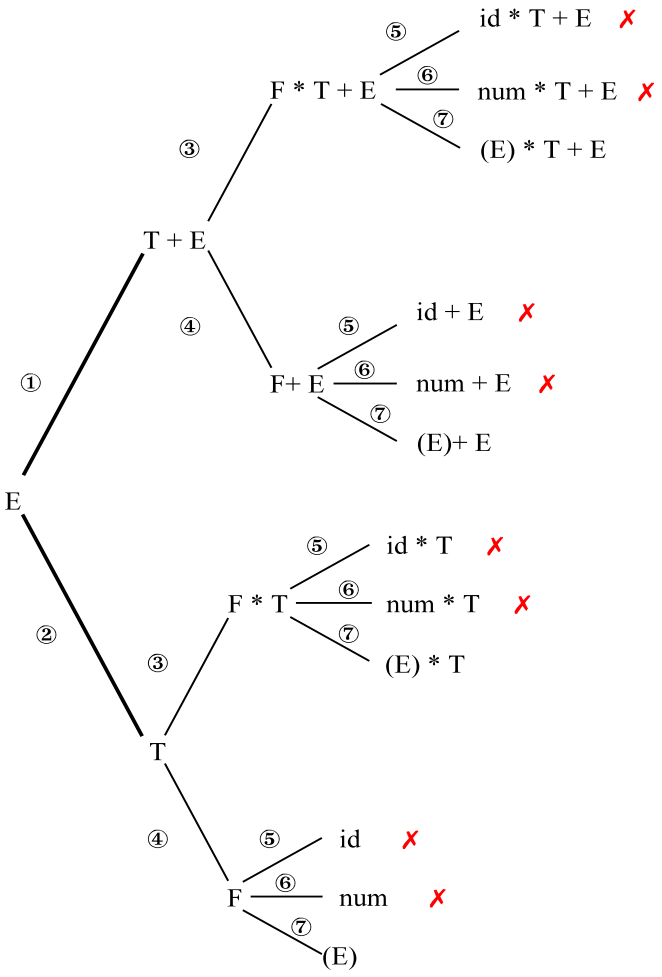


Figura 3.4. Árbol universal de la gramática del cuadro 3.3. Sólo se han representado las derivaciones a izquierda y los cuatro primeros niveles del árbol

con la cabeza de la secuencia a reconocer. Pueden establecerse otros **criterios de poda**, pero éste es el que utilizaremos en nuestro estudio.

Resumiendo, se pretende recorrer el árbol universal profundizando por cada rama hasta llegar a encontrar una forma sentencial que no puede coincidir con la sentencia que se busca, en cuyo caso se desecha la rama; o que coincide con lo buscado, momento en que se acepta la sentencia. Si por ninguna rama se puede reconocer, se rechaza la sentencia.

En el árbol universal de la figura 3.4 se han marcado de rojo las ramas por las que ya no tiene sentido continuar en base a nuestro criterio de poda. Por el resto de ramas proseguiría la búsqueda de la sentencia "(id + num) * id + num". Además, cada eje se ha numerado con el identificador de la regla que se ha aplicado.

Con el criterio de poda escogido la búsqueda en profundidad no funcionará si la gramática es recursiva a la izquierda, ya que existirán ramas infinitas en las que el número de terminales a la izquierda de una forma sentencial no aumenta y nuestra búsqueda se meterá en una recursión infinita. En estos casos es necesario incluir nuevos criterios de poda.

Los analizadores sintácticos con retroceso no suelen utilizarse en la práctica. Esto se debe a que el retroceso es sumamente ineficiente y existen otros tipos de análisis más potentes basados en mecanismos diferentes, como veremos más adelante. Incluso para el lenguaje natural el retroceso no es nada eficiente, y se prefieren otros métodos.

A continuación se presenta el algoritmo de análisis descendente con retroceso:

```
Precondición: formaSentencial ≡ μ A δ, con μ ∈ T*, A ∈ N, y δ ∈ (N ∪ T)*
AnálisisDescendenteConRetroceso(formaSentencial, entrada)
    Si μ ≠ parteIzquierda(entrada) Retornar Fin Si
    Para cada  p_i ∈ {p_i ∈ P / p_i ≡ A → α_i}
        formaSentencial´≡ μ α_i δ ≡ μ´ A´ δ´,        con μ´ ∈ T*, A´ ∈ N,
                                                    y δ´ ∈ (N ∪ T)*
        Si μ´ ≡ entrada
            ¡ Sentencia reconocida !
            Retornar
        Fin Si
        AnálisisDescendenteConRetroceso(formaSentencial´, entrada)
        Si ¡ Sentencia reconocida ! Retornar Fin Si
    Fin For
Fin AnálisisDescendenteConRetroceso
```

Y la llamada principal quedaría:

```
AnálisisDescendenteConRetroceso(axiomaInicial, cadenaAReconocer)
Si NOT ¡ Sentencia reconocida !
        ¡¡ Sentencia no reconocida !!
Fin Si
```

La tabla siguiente muestra la ejecución de este algoritmo para reconocer o rechazar la sentencia "(id + num) * id + num". Se aplican siempre derivaciones a izquierda, y la columna "Pila de reglas utilizadas" representa la rama del árbol universal sobre la que se está trabajando. Se han representado en rojo los retrocesos. Nótese como la tabla es mucho más extensa de lo que aquí se ha representado, lo que da una estimación de la ineficiencia del algoritmo.

Forma sentencial	Pila de reglas utilizadas
E	1
T + E	1-3
F * T +E	1-3-5
id * T + E	1-3
F * T +E	1-3-6
num * T + E	1-3
F * T +E	1-3-7
(E) * T +E	1-3-7-1
(T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-5
(id * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-6
(num * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-7
((E) * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1
(T + E) * T + E	1-3-7-1-4
(F + E) * T + E	1-3-7-1-4-5
(id + E) * T + E	1-3-7-1-4-5-1
(id + T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1-3-5
(id + id * T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1-3-6
(id + num * T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1-3-7
(id + (E) * T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1
(id + T + E) * T + E	1-3-7-1-4-5-1-4
(id + F + E) * T + E	
....	
(id + E) * T + E	1-3-7-1-4-5-2
(id + T) * T + E	
...	

3.5.2 Análisis descendente con funciones recursivas

Una gramática de contexto libre constituye un mecanismo para expresar un lenguaje formal. Sin embargo no es el único, ya que la notación BNF y los diagramas de sintaxis tienen la misma potencia.

Partiendo de que la notación BNF también permite expresar un lenguaje formal, a continuación se dará una definición de diagrama de sintaxis y se demostrará que éstos tienen, como mínimo, la misma potencia que la mencionada notación BNF. A partir de este punto, seremos capaces de construir *ad hoc* un analizador sintáctico mediante cualquier lenguaje de programación que permita la recursión.

3.5.2.1 Diagramas de sintaxis

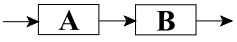
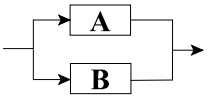
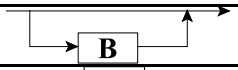
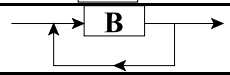
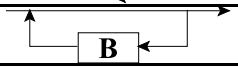
Un diagrama de sintaxis (también llamados diagramas de Conway) es un grafo dirigido donde los elementos no terminales aparecen como rectángulos, y los terminales como círculos o elipses.

Todo diagrama de sintaxis posee un origen y un destino, que no se suelen representar explícitamente, sino que se asume que el origen se encuentra a la izquierda del diagrama y el destino a la derecha.

Cada arco con origen en α y destino en β representa que el símbolo α puede ir seguido del β (pudiendo ser α y β tanto terminales como no terminales). De esta forma todos los posibles caminos desde el inicio del grafo hasta el final, representan formas sentenciales válidas.

3.5.2.2 Potencia de los diagramas de sintaxis

Demostraremos que los diagramas de sintaxis permiten representar las mismas gramáticas que la notación BNF, por inducción sobre las operaciones básicas de BNF:

Operación	BNF	Diagrama de sintaxis
Yuxtaposición	AB	
Opción	A | B	
	ε | B	
Repetición	1 o más veces {B}	
	0 o más veces [B]	

Siguiendo la «piedra de Rosetta» que supone la tabla anterior, es posible convertir cualquier expresión BNF en su correspondiente diagrama de sintaxis, ya que A y B pueden ser, a su vez, expresiones o diagramas complejos.

Por otro lado, quien haya trabajado con profusión los diagramas de sintaxis

habrá observado que no siempre resulta fácil convertir un diagrama de sintaxis cualquiera a su correspondiente en notación BNF.

3.5.2.3 Correspondencia con flujos de ejecución

Ahora vamos a establecer una correspondencia entre el flujo de ejecución de un programa y el camino que se puede seguir en uno o varios diagramas de sintaxis para reconocer una sentencia válida. Por ejemplo, la yuxtaposición quedaría como ilustra la figura 3.10.

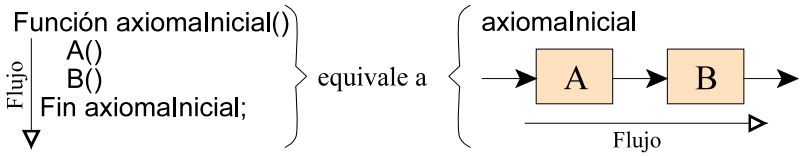


Figura 3.10. Equivalencia entre el diagrama de sintaxis de yuxtaposición y una función que sigue el mismo flujo. La función se llama igual que el diagrama.

Flujo

Es importante el ejercicio de programación consistente en seguir la evolución detallada de las llamadas de los procedimientos entre sí. Antes de comenzar con un ejemplo completo conviene reflexionar sobre cómo actuar en caso de que nos encontremos con un símbolo terminal en el diagrama de sintaxis. En tales situaciones habrá que controlar que el *token* que nos devuelve el analizador léxico coincide con el que espera el diagrama de sintaxis. Por ejemplo, el diagrama de la figura 3.11 se traduciría por el bloque de código:

```
Función prog ( )
        sent ( );
```

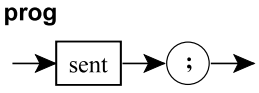


Figura 3.11. Un diagrama de sintaxis para prog

```
        Si el siguiente token es ';' entonces
                consumir el token
        si no
                ¡Error sintáctico!
        Fin si
Fin prog
```

donde consumir el token hace referencia a que se le pide el siguiente *token* al analizador léxico. Para el resto de ejemplos supondremos que el analizador léxico suministra una función denominada get_token() que almacena el siguiente *token* en la variable global token. Siguiendo este criterio, si partiéramos del diagrama de la figura 3.12 el código generado sería:

```
Función prog2 ( );
        get_token()
```

```
        sent ( );
        Si token == ';' entonces
                get_token()
        si no
                ¡Error sintáctico!
        Fin si
        Si token == FIN entonces
                get_token()
        si no
                ¡Error sintáctico!
        Fin si
Fin Prog2
```

Es importante darse cuenta de que para que el analizador sintáctico pueda comenzar el reconocimiento, la variable token debe tener ya un valor pre-cargado; a esta variable se la llama *token* de prebúsqueda (*lookahead*), y es de fundamental importancia, puesto que es la que nos permitirá tomar la decisión de por dónde debe continuar el flujo en el diagrama de sintaxis, como veremos más adelante. También es importante apreciar que lo que aquí se ha denotado genéricamente por ¡Error sintáctico! consiste realmente en un mensaje indicativo del número de línea en que se ha encontrado el error, así como su naturaleza, p.ej.: «Se esperaba una ',' y se ha encontrado token»

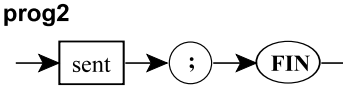


Figura 3.12. Un nuevo diagrama de sintaxis para prog: prog2

3.5.2.4 Ejemplo completo

En este apartado vamos a construir las funciones recursivas para una serie de diagramas que representan un lenguaje consistente en expresiones lógico/aritméticas finalizada cada una de ellas en punto y coma. El axioma inicial viene dado por el no terminal secuencia, quien también debe encargarse de hacer el primer get_token(), así como de controlar que el último *pseudotoken* enviado por el analizador léxico sea **EOF** (*End Of File*).

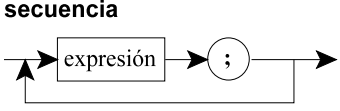


Figura 3.13. Diagrama de sintaxis de secuencia

```
Función secuencia ( ) { // Figura 3.13
        get_token ( );
        do {
            expresión ( );
            while (token != PUNTOYCOMA) {
```

```
                    !Error en expresión¡
                    get_token ( );
                };
                get_token( );
            } while (token != EOF);
};
```

En este caso se considera al ';' (PUNTOYCOMA) como un *token* de seguridad, lo que permite hacer una recuperación de errores mediante el método *panic mode* (ver punto 3.3.1).

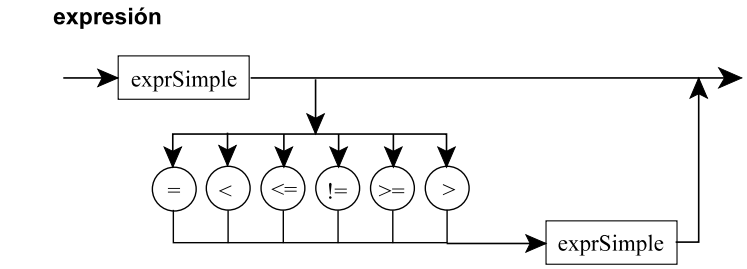


Figura 3.14. Diagrama de sintaxis de expresión

```
Función expresión() { // Figura 3.14
        exprSimple();
        if ((token == IGUAL)|| (token == ME)|| (token == MEI)||
           (token == DIST)|| (token == MAI)|| (token == MA)) {
                get_token();
                exprSimple();

        }
}
```

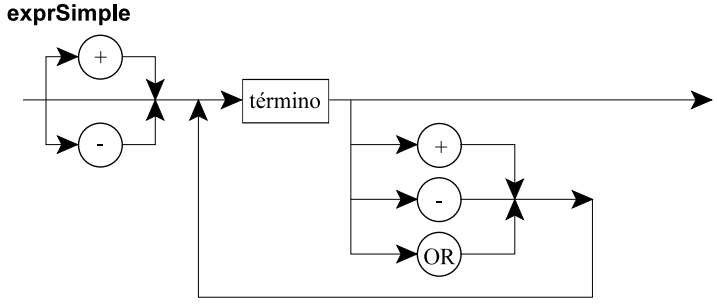


Figura 3.15. Diagrama de sintaxis de exprSimple

```
Función exprSimple ( ) { // Figura 3.15
        if ((token == MAS) || (token == MENOS)) {
           get_token( );
        }
```

```
            término ( );
            while ((token == MAS) || (token == MENOS) || (token == OR)) {
              get_token( );
              término ( );
            }
}
```

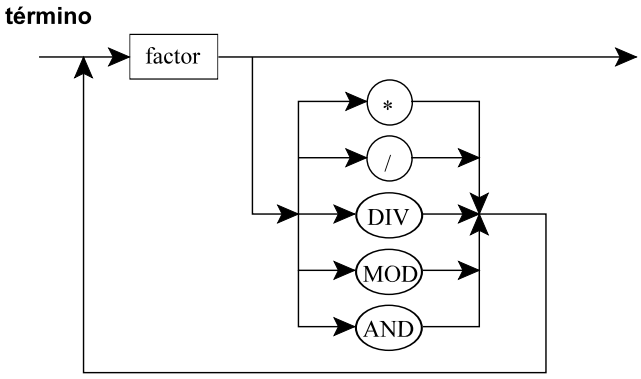


Figura 3.16. Diagrama de sintaxis de término

```
Función término() { // Figura 3.16
            factor();
            while ((token == POR) || (token == DIV) || (token == DIV_ENT) ||
                    (token == MOD) || (token == AND) {
                        get_token();
                        factor();
            }
}
```

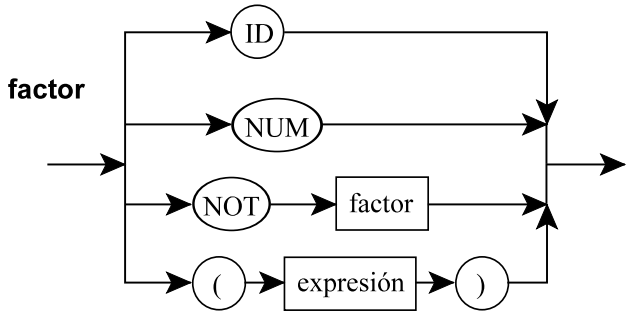


Figura 3.17. Diagrama de sintaxis de factor

```
Función factor ( ){ // Figura 3.17
            switch (token) {
```

```
                case ID : get_token(); break;
                case NUM : get_token(); break;
                case NOT : get_token(); factor(); break;
                case AB_PARID :     get_token();
                                    expresión();
                                    if (token != CE_PAR) {
                                       Error: Paréntesis de cierre
                                    } else get_token();
                                    break;
                default : Error: Expresión no válida.
            }
}
```

Este ejemplo muestra cómo resulta relativamente fácil obtener funciones recursivas que simulen en ejecución el camino seguido por un diagrama de sintaxis para reconocer una cadena.

3.5.2.5 Conclusiones sobre el análisis descendente con funciones recursivas

No todo diagrama de sintaxis es susceptible de ser convertido tan fácilmente en función recursiva; concretamente, en aquellos nudos del diagrama en los que el flujo puede seguir por varios caminos, la decisión sobre cuál seguir se ha hecho en base al siguiente *token* de la entrada, de manera que dicho *token* actúa como discriminante. El diagrama de la figura 3.18.a ilustra un ejemplo en el cual necesitamos más de un *token* (concretamente dos) para tomar correctamente la decisión de qué camino seguir. Por

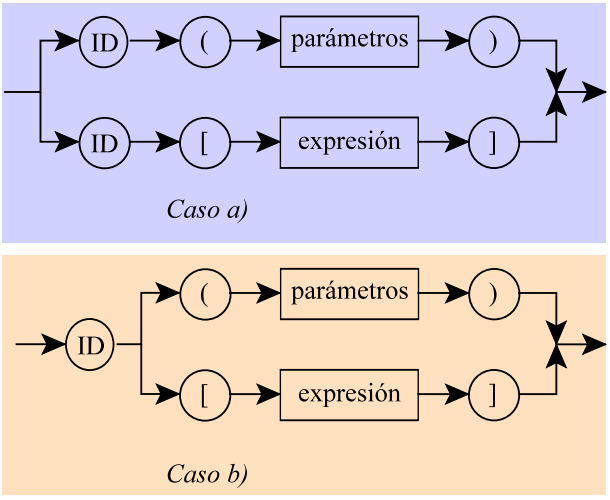


Figura 3.18. Diagramas de sintaxis equivalentes. El caso a) tiene dos alternativas que comienzan por el mismo *token*. El caso b) ha eliminado este problema

regla general, estos diagramas pueden reconstruirse de manera que se adapten bien a nuestro mecanismo de construcción de funciones, tal y como ilustra la figura 3.18.b.

El metacompilador JavaCC utiliza la notación BNF para expresar una gramática, y construye una función recursiva por cada no terminal de la gramática incluyendo en ella toda la lógica interna de reconocimiento. En los casos de conflicto, como el ilustrado anteriormente, el desarrollador del compilador puede darle a JavaCC una indicación para que utilice un mayor número de *tokens* de pre-búsqueda.

3.5.3 Análisis descendente de gramáticas LL(1)

Este tipo de análisis se basa en un autómata de reconocimiento en forma de tabla, denominada tabla de chequeo de sintaxis. Dicha tabla posee como eje de ordenadas a los no terminales de la gramática, y como abcisas a los terminales (incluido en *pseudotoken* EOF). El contenido de cada casilla interior (que se corresponde con la columna de un terminal y la fila de un no terminal) contiene, o bien una regla de producción, o bien está vacía, lo que se interpreta como un rechazo de la cadena a reconocer.

En general, podemos decir que una gramática LL(1) es aquélla en la que es suficiente con examinar sólo un símbolo a la entrada, para saber qué regla aplicar, partiendo del axioma inicial y realizando derivaciones a izquierda. Toda gramática reconocible mediante el método de los diagramas de sintaxis siguiendo el procedimiento visto en el punto anterior es LL(1).

El método consiste en seguir un algoritmo partiendo de:

- La cadena a reconocer, junto con un apuntador, que nos indica cual es el *token* de pre-búsqueda actual; denotaremos por **a** dicho *token*.
- Una pila de símbolos (terminales y no terminales); denotaremos por **X** la cima de esta pila.
- Una tabla de chequeo de sintaxis asociada de forma unívoca a una gramática. En el presente texto no se trata la forma en que dicha tabla se obtiene a partir de la gramática. Denotaremos por **M** a dicha tabla, que tendrá una fila por cada no terminal de la gramática de partida, y una columna por cada terminal incluido el EOF: $M[N \times T \cup \{\$\}]$.

Como siempre, la cadena de entrada acabará en EOF que, a partir de ahora, denotaremos por el símbolo '$'. El mencionado algoritmo consiste en consultar reiteradamente la tabla **M** hasta aceptar o rechazar la sentencia. Partiendo de una pila que posee el $ en su base y el axioma inicial S de la gramática encima, cada paso de consulta consiste en seguir uno de los puntos siguientes (son excluyentes):

1.- Si $X = a = \$$ entonces ACEPTAR.

2.- Si $X = a \neq \$$ entonces
- se quita X de la pila
- y se avanza el apuntador.

3.- Si X ∈ T y X ≠ a entonces RECHAZAR.

4.- Si X ∈ N entonces consultamos la entrada M[X,a] de la tabla, y :
- Si M[X,a] es vacia : RECHAZAR.
- Si M [X,a] no es vacia, se quita a X de la pila y se inserta el consecuente en orden inverso. Ejemplo: Si M[X,a] = {X → UVY}, se quita a X de la pila, y se meten UVY en orden inverso, como muestra la figura 3.19.

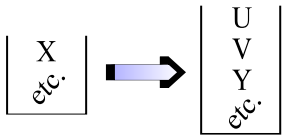


Figura 3.19. Aplicación de regla en el método LL(1)

Al igual que sucedía en el análisis descendente con funciones recursivas, una vez aplicada una regla, ésta ya no será desaplicada por ningún tipo de retroceso.

Uno de los principales inconvenientes de este tipo de análisis es que el número de gramáticas LL(1) es relativamente reducido; sin embargo, cuando una gramática no es LL(1), suele ser posible traducirla para obtener una equivalente que sí sea LL(1), tras un adecuado estudio. Por ejemplo, la siguiente gramática no es LL(1):

E → E + T | T
T → T * F | F
F → (E) | id

pero puede ser factorizada, lo que la convierte en:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

que es equivalente y LL(1), siendo su tabla M:

T / N	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → id			F → (E)		

Evidentemente, esta tabla pertenece a la gramática y puede emplearse para

rechazar o reconocer cualesquiera sentencias, esto es, la tabla no se construye para cada sentencia a reconocer, sino que está asociada exclusivamente a la gramática y viene a ser una representación de su autómata finito con pila.

La secuencia siguiente muestra la ejecución de este algoritmo para reconocer o rechazar la sentencia "id * (id + id) $". La columna "Acción" indica qué punto del algoritmo es el que se aplica, mientras que la columna "Cadena de Entrada" indica, a la izquierda, qué *tokens* se han consumido, a la derecha lo que quedan por consumir y, subrayado, el *token* de pre-búsqueda actual. Puede observarse cómo la ejecución de este método es mucho más eficiente que el caso con retroceso, e igual de eficiente que el análisis con funciones recursivas.

Pila de símbolos	Cadena de Entrada	Acción
$ E	id * (id + id) $	4.- M[E, id] = E → T E'
$ E' T	id * (id + id) $	4.- M[T, id] = T → F T'
$ E' T' F	id * (id + id) $	4.- M[F, id] = F → id
$ E' T' id	id * (id + id) $	2.- id = id
$ E' T'	id * (id + id) $	4.- M[T', *] = T' → * F T'
$ E' T' F *	id * (id + id) $	2.- * = *
$ E' T' F	id * (id + id) $	4.- M[F, (] = F → (E)
$ E' T') E (	id * (id + id) $	2.- (= (
$ E' T') E	id * (id + id) $	4.- M[E, id] = E → T E'
$ E' T') E' T	id * (id + id) $	4.- M[T, id] = T → F T'
$ E' T') E' T' F	id * (id + id) $	4.- M[F, id] = F →id
$ E' T') E' T' id	id * (id + id) $	2.- id = id
$ E' T') E' T'	id * (id + id) $	4.- M[T',+] = T' → ε
$ E' T') E'	id * (id + id) $	4.- M[E',+] = E' →+ T E'
$ E' T') E' T +	id * (id + id) $	2.- + = +
$ E' T') E' T	id * (id + id) $	4.- M[T, id] = T →F T'
$ E' T') E' T' F	id * (id + id) $	4.- M[F, id] = F →id
$ E' T') E' T' id	id * (id + id) $	2.- id = id
$ E' T') E' T'	id * (id + id) $	4.- M[T',)] = T' → ε
$ E' T') E'	id * (id + id) $	4.- M[E',)] = E' → ε
$ E' T')	id * (id + id) $	2.-) =)
$ E' T'	id * (id + id) $	4.- M[T', $] = T' → ε
$ E'	id * (id + id) $	4.- M[E', $] = E' → ε
$	id * (id + id) $	1.- $ = $ ACEPTAR

La secuencia de reglas aplicadas (secuencia de pasos de tipo 4) proporciona la secuencia de derivaciones a izquierda que representa el árbol sintáctico que reconoce la sentencia.

Este método tiene una ventaja sobre el análisis con funciones recursivas, y es que la construcción de la tabla de chequeo de sintaxis puede automatizarse, por lo que una modificación en la gramática de entrada no supone un problema grande en lo que a reconstrucción de la tabla se refiere. No sucede lo mismo con las funciones

recursivas, ya que estas tienen que ser reconstruidas a mano con la consecuente pérdida de tiempo y propensión a errores.

3.5.4 Generalidades del análisis ascendente

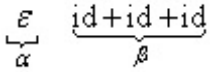
Antes de particularizar en los distintos tipos de análisis ascendentes, resulta conveniente estudiar los conceptos y metodología comunes a todos ellos. El objetivo de un análisis ascendente consiste en construir el árbol sintáctico desde abajo hacia arriba, esto es, desde los *tokens* hacia el axioma inicial, lo cual disminuye el número de reglas mal aplicadas con respecto al caso descendente (si hablamos del caso con retroceso) o amplia el número de gramáticas susceptibles de ser analizadas (si hablamos del caso LL(1)).

Tanto si hay retroceso como si no, en un momento dado, la cadena de entrada estará dividida en dos partes, denominadas α y β:

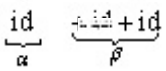
β: representa el trozo de la cadena de entrada (secuencia de *tokens*) por consumir: $\beta \in T^*$. Coincidirá siempre con algún trozo de la parte derecha de la cadena de entrada. Como puede suponerse, inicialmente β coincide con la cadena a reconocer al completo (incluido el EOF del final).

α: coincidirá siempre con el resto de la cadena de entrada, trozo al que se habrán aplicado algunas reglas de producción en sentido inverso: $\alpha \in (N \cup T)^*$.

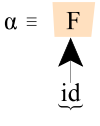
Por ejemplo, si quisiéramos reconocer "id + id + id", partiendo de la gramática del cuadro 3.3 se comenzaría a construir el árbol sintáctico a partir del árbol vacío (α = ε) y con toda la cadena de entrada por consumir (β = id + id + id):



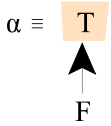
, y tras consumir el primer *token*:



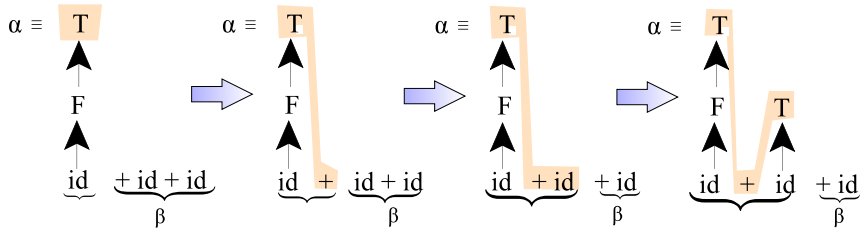
, y ahora podemos aplicar la regla ⑤ hacia atrás, con lo que α es F:



, a lo que ahora se puede aplicar la regla ④, produciendo:



. Así, poco a poco se va construyendo el árbol:



Como puede verse, α representa al árbol sintáctico visto desde arriba conforme se va construyendo ascendentemente: α es una forma sentencial ascendente.

3.5.4.1 Operaciones en un analizador ascendente

A medida que un analizador sintáctico va construyendo el árbol, se enfrenta a una configuración distinta (se denomina configuración al par α-β) y debe tomar una decisión sobre el siguiente paso u operación a realizar. Básicamente se dispone de cuatro operaciones diferentes, y cada tipo de analizador ascendente se distingue de los demás en base a la inteligencia sobre cuándo aplicar cada una de dichas operaciones. Cualquier mecanismo de análisis ascendente consiste en partir de una configuración inicial e ir aplicando operaciones, cada una de las cuales permite pasar de una configuración origen a otra destino. El proceso finalizará cuando la configuración destino llegue a ser tal que α represente al árbol sintáctico completo y en β se hayan consumido todos los *tokens*. Las operaciones disponibles son las siguientes:

1.- ACEPTAR: se acepta la cadena: β ≡ EOF y α ≡ S (axioma inicial).

2.- RECHAZAR: la cadena de entrada no es válida.

3.- **Reducir**: consiste en aplicar una regla de producción hacia atrás a algunos elementos situados en el extremo derecho de α. Por ejemplo, si tenemos la configuración:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ X_{p+1}\ \ldots\ X_m]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n]$$

(recordemos que $X_i \in (N \cup T)^*$ y $a_i \in T$), y existe una regla de la forma

$$A_k \rightarrow X_{p+1}\ \ldots\ X_m$$

entonces una reducción por dicha regla consiste en pasar a la configuración:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ A_k]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n].$$

Resulta factible reducir por reglas épsilon, esto es, si partimos de la configuración:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ X_{p+1}\ \ldots\ X_m]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n]$$

puede reducirse por una regla de la forma:

$$A_k \rightarrow \varepsilon$$

obteniéndose:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ X_{p+1}\ \ldots\ X_m\ A_k]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n]$$

4.- **Desplazar**: consiste únicamente en quitar el terminal más a la izquierda de β y ponerlo a la derecha de α. Por ejemplo, si tenemos la configuración:

$$\alpha \equiv [X_1\ X_2\ ...\ X_p\ X_{p+1}\ ...\ X_m]\text{-}\beta \equiv [a_{m+1}\ a_{m+2}\ ...\ a_n]$$

un desplazamiento pasaría a:

$$\alpha \equiv [X_1\ X_2\ ...\ X_p\ X_{p+1}\ ...\ X_m\ a_{m+1}]\text{-}\beta \equiv [a_{m+2}\ ...\ a_n]$$

Resumiendo, mediante reducciones y desplazamientos, tenemos que llegar a aceptar o rechazar la cadena de entrada. Antes de hacer los desplazamientos tenemos que hacerles todas las reducciones posibles a α, puesto que éstas se hacen a la parte derecha de α, y no por enmedio. Cuando α es el axioma inicial y β es la tira nula (sólo contiene EOF), se acepta la cadena de entrada. Cuando β no es la tira nula o α no es el axioma inicial y no se puede aplicar ninguna regla, entonces se rechaza la cadena de entrada.

3.5.5 Análisis ascendente con retroceso

Al igual que ocurría con el caso descendente, este tipo de análisis intenta probar todas las posibles operaciones (reducciones y desplazamientos) mediante un método de fuerza bruta, hasta llegar al árbol sintáctico, o bien agotar todas las opciones, en cuyo caso la cadena se rechaza. El algoritmo es el siguiente:

```
Precondición: α ∈ (N ∪ T)* y β ∈ T*
AnálisisAscendenteConRetroceso(α, β)
    Para cada p_i ∈ P hacer
        Si consecuente(p_i) ≡ cola(α)
            α'-β' = reducir α-β por la regla p_i
            Si α ≡ S AND β≡ ε
                ¡ Sentencia reconocida ! (ACEPTAR)
            si no
                AnálisisAscendenteConRetroceso(α', β')
            Fin si
        Fin si
    Fin para
    Si β ≠ ε
        α'-β' = desplazar α-β
        AnálisisAscendenteConRetroceso(α', β')
    Fin si
Fin AnálisisAscendenteConRetroceso
```

Y la llamada principal quedaría:

```
AnálisisAscendenteConRetroceso(ε, cadenaAReconocer)
Si NOT ¡ Sentencia reconocida !
        ¡¡ Sentencia no reconocida !! (RECHAZAR)
Fin Si
```

Como el lector habrá podido apreciar, este algoritmo incurre en recursión sin fin ante gramáticas que posean reglas épsilon (esto es, que permitan la cadena vacía como parte del lenguaje a reconocer). Esto se debe a que una regla épsilon siempre puede aplicarse hacia atrás, dando lugar a una rama infinita hacia arriba. P.ej., sea cual sea la cadena a reconocer, si existe una regla épsilon $F \rightarrow \varepsilon$ se obtendrá mediante este

algoritmo algo parecido a la figura 3.24

Retomemos a continuación la gramática del cuadro 3.1 y veamos el proceso que se sigue para reconocer la cadena “id * id”:

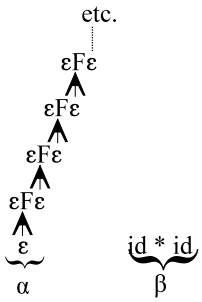


Figura 3.24. Nótese como la cola de α puede ser ε, Fε, εFε, etc., lo que hace que la regla F → ε pueda aplicarse indefinidamente.

Pila de reglas utilizadas	α	β	Acción
--	ε	id * id	Desplazar
--	id	* id	Reducir por F → id
5	F	* id	Reducir por T → F
5-4	T	* id	Reducir por E → T
5-4-2	E	* id	Desplazar
5-4-2	E *	id	Desplazar
5-4-2	E * id	ε	Reducir por F → id
5-4-2-5	E * F	ε	Reducir por T → F
5-4-2-5-4	E * T	ε	Reducir por E → T
5-4-2-5-4-2	E * E	ε	Retroceso (pues no hay nada por desplazar)
5-4-2-5-4	E * T	ε	Retroceso (pues no hay nada por desplazar)
5-4-2-5	E * F	ε	Retroceso (pues no hay nada por desplazar)
5-4-2	E * id	ε	Retroceso (pues no hay nada por desplazar)
5-4-2	E *	id	Retroceso (pues ya se desplazó)
5-4-2	E	* id	Retroceso (pues ya se desplazó)
5-4	T	* id	Desplazar
5-4	T *	id	Desplazar
5-4	T * id	ε	Reducir por F → id
5-4-5	T * F	ε	Reducir por T → T * F
5-4-5-3	T	ε	Reducir por E → T
5-4-5-3-2	E	ε	Aceptar

Este método es más eficiente que el descendente con retroceso puesto que consume los *tokens* a mayor velocidad y, por tanto, trabaja con mayor cantidad de información a la hora de tomar cada decisión. No obstante resulta inviable para aplicaciones prácticas pues su ineficiencia sigue siendo inadmisible.

3.5.6 Análisis ascendente de gramáticas LR(1)

En este epígrafe se introducirá una técnica eficiente de análisis sintáctico ascendente que se puede utilizar para procesar una amplia clase de gramáticas de contexto libre. La técnica se denomina análisis sintáctico LR(k). La abreviatura LR obedece a que la cadena de entrada es examinada de izquierda a derecha (en inglés, *Left-to-right*), mientras que la "R" indica que el proceso proporciona el árbol sintáctico mediante la secuencia de derivaciones a derecha (en inglés, *Rightmost derivation*) en orden inverso. Por último, la "k" hace referencia al número de *tokens* de pre-búsqueda utilizados para tomar las decisiones sobre si reducir o desplazar. Cuando se omite, se asume que k, es 1.

El análisis LR es atractivo por varias razones.

- Pueden reconocer la inmensa mayoría de los lenguajes de programación que puedan ser generados mediante gramáticas de contexto-libre.
- El método de funcionamiento de estos analizadores posee la ventaja de localizar un error sintáctico casi en el mismo instante en que se produce con lo que se adquiere una gran eficiencia de tiempo de compilación frente a procedimientos menos adecuados como puedan ser los de retroceso. Además, los mensajes de error indican con precisión la fuente del error.

El principal inconveniente del método es que supone demasiado trabajo construir manualmente un analizador sintáctico LR para una gramática de un lenguaje de programación típico, siendo necesario utilizar una herramienta especializada para ello: un generador automático de analizadores sintácticos LR. Por fortuna, existen disponibles varios de estos generadores. Más adelante estudiaremos el diseño y uso de algunos de ellos: los metacompiladores Yacc, Cup y JavaCC. Con Yacc o Cup se puede escribir una gramática de contexto libre generándose automáticamente su analizador sintáctico. Si la gramática contiene ambigüedades u otras construcciones difíciles de analizar en un examen de izquierda a derecha de la entrada, el metacompilador puede localizar las reglas de producción problemáticas e informar de ellas al diseñador.

Esta técnica, al igual que la del LL(1), basa su funcionamiento en la existencia de una tabla especial asociada de forma única a una gramática. Existen varias técnicas para construir dicha tabla, y cada una de ellas produce un "sucedáneo" del método principal. La mayoría de autores se centra en tres métodos principales. El primero de ellos, llamado LR sencillo (*SLR*, en inglés) es el más fácil de aplicar, pero el menos poderoso de los tres ya que puede que no consiga producir una tabla de análisis sintáctico para algunas gramáticas que otros métodos sí consiguen. El segundo método, llamado LR canónico, es el más potente pero el más costoso. El tercer método, llamado LR con examen por anticipado (*Look Ahead LR*, en inglés), está entre los otros dos en cuanto a potencia y coste. El método LALR funciona con las gramáticas de la mayoría de los lenguajes de programación y, con muy poco esfuerzo, se puede implantar de forma eficiente. Básicamente, un método es más potente que otro en función del número de gramáticas a que puede aplicarse. Mientras más amplio sea el espectro de gramáticas admitidas más complejo se vuelve el método.

Funcionalmente hablando, un analizador LR consta de dos partes bien diferenciadas: a) un programa de proceso y b) una tabla de análisis. El programa de proceso posee, como se verá seguidamente, un funcionamiento muy simple y permanece invariable de analizador a analizador. Según sea la gramática a procesar deberá variarse el contenido de la tabla de análisis que es la que identifica plenamente al analizador.

La figura 3.25 muestra un esquema sinóptico de la estructura general de una analizador LR. Como puede apreciarse en ella, el analizador procesa una cadena de entrada finalizada con el símbolo $ que representa el delimitador EOF. Esta cadena se lee de izquierda a derecha, y el reconocimiento de un solo símbolo permite tomar las decisiones oportunas (LR(1)).

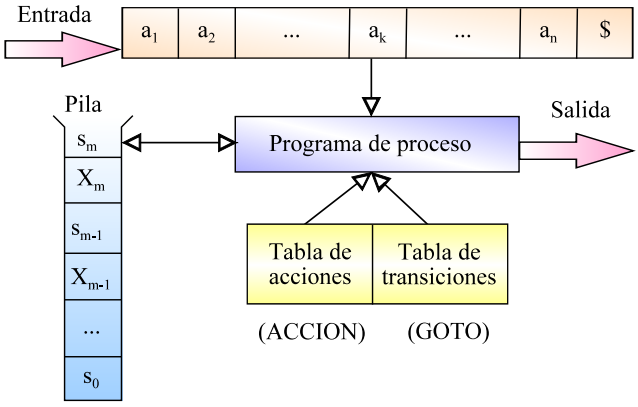


Figura 3.25. Esquema de funcionamiento de un analizador LR cualquiera

Además, el algoritmo hace uso de una pila que tiene la forma:

$$s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m$$

donde el símbolo s_m se encuentra en la cima tal y como se muestra en la figura 3.25. Cada uno de los X_i es un símbolo de la gramática ($X_i \in (N \cup T)^*$) y los s_i representan distintos estados del autómata asociado a la gramática; al conjunto de estados lo denominaremos E: $s_i \in E$. Cada estado s_i tiene asociado un símbolo X_i, excepto el s_0 que representa al estado inicial y que no tiene asociado símbolo ninguno. Los estados se utilizan para representar toda la información contenida en la pila y situada antes del propio estado. Consultando el estado en cabeza de la pila y el siguiente *token* a la entrada se decide qué reducción ha de efectuarse o bien si hay que desplazar.

Por regla general una tabla de análisis para un reconocedor LR consta de dos partes claramente diferenciadas entre sí que representan dos tareas distintas, la tarea de transitar a otro estado (GOTO) y la tarea de qué acción realizar (ACCION). La tabla GOTO es de la forma $E \times (N \cup T \cup \{\$\})$ y contiene estado de E, mientras que la tabla

ACCION es de la forma $E \times (N \cup T)$ y contiene una de las cuatro acciones que vimos en el apartado 3.5.4.

Suponiendo que en un momento dado el estado que hay en la cima de la pila es s_m y el *token* actual es a_i, el funcionamiento del analizador LR consiste en aplicar reiteradamente los siguientes pasos, hasta aceptar o rechazar la cadena de entrada:

1.- Consultar la entrada (s_m, a_i) en la tabla de ACCION. El contenido de la casilla puede ser:

$$\text{ACCION}(s_m, a_i) \equiv \begin{cases} \text{Aceptar} \\ \text{Rechazar} \\ \text{Reducir por } A \rightarrow \tau \\ \text{Desplazar} \end{cases}$$

Si se acepta o rechaza, se da por finalizado el análisis, si se desplaza se mete a_i en la pila, y si se reduce, entonces la cima de pila coincide con el consecuente τ (amén de los estados), y se sustituye τ por A.

2.- Una vez hecho el proceso anterior, en la cima de la pila hay un símbolo y un estado, por este orden, lo que nos dará una entrada en la tabla de GOTO. El contenido de dicha entrada se coloca en la cima de la pila.

Formalmente se define una **configuración de un analizador LR** como un par de la forma:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m \ , \ a_i a_{i+1} \ldots a_n \$)$$

es decir, el primer componente es el contenido actual de la pila, y el segundo la subcadena de entrada que resta por reconocer, siendo a_i el *token* de pre-búsqueda. Partiremos de esta configuración general para el estudio que prosigue.

Así, formalmente, cada ciclo del algoritmo LR se describe como:

1.1.- Si ACCION(s_m, a_i) = Desplazar, entonces se introduce en la pila el símbolo a_i, produciendo:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_i \ , \ a_{i+1} \ldots a_n \$)$$

A continuación se consulta la entrada (s_m, a_i) de la tabla GOTO: GOTO$(s_m, a_i) = s_{m+1}$, produciéndose finalmente:

$$(s_0 X_1 s_1 X2 s_2 \ldots X_m s_m a_i \ s_{m+1} \ , \ a_{i+1} \ldots a_n \$)$$

pasando s_{m+1} a estar situado en cabeza de la pila y a_{i+1} el siguiente símbolo a explorar en la cinta de entrada.

1.2.- Si ACCION(s_m, a_i) = Reducir por $A \rightarrow \tau$, entonces el analizador ejecuta la reducción oportuna donde el nuevo estado en cabeza de la pila se obtiene mediante la función GOTO$(s_{m-r}, a_i) = s$ donde r es precisamente la longitud del consecuente τ. O sea, el analizador extrae primero $2 \cdot r$ símbolos de la pila (r estados y los r símbolos de la gramática que cada uno tiene por debajo), exponiendo el estado s_{m-r} en la cima. Luego se introduce el no terminal A

(antecedente de la regla aplicada), produciendo:

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_{m-r}\ s_{m-r}\ A \quad , \quad a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

A continuación se consulta la entrada (s_{m-r}, a_i) de la tabla GOTO: GOTO(s_{m-r}, A) = s_{m-r+1}, produciéndose finalmente:

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_{m-r}\ s_{m-r}\ A\ s_{m-r+1} \quad , \quad a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

donde s_{m-r+1} es el nuevo estado en la cima de la pila y no se ha producido variación en la subcadena de entrada que aún queda por analizar.

1.3.- Si ACCION(s_m, a_i) = ACEPTAR, entonces se ha llegado a la finalización en el proceso de reconocimiento y el análisis termina aceptando la cadena de entrada.

1.4-. Si ACCION(s_m, a_i) = RECHAZAR, entonces el analizador LR ha descubierto un error sintáctico y se debería proceder en consecuencia activando las rutinas de recuperación de errores. Como se ha comentado, una de las ventajas de este tipo de análisis es que cuando se detecta un error, el *token* erróneo suele estar al final de α o al principio de β, lo que permite depurar con cierta facilidad las cadenas de entrada (programas).

La aplicación de uno de estos cuatro pasos se aplica de manera reiterada hasta aceptar o rechazar la cadena (en caso de rechazo es posible continuar el reconocimiento si se aplica alguna técnica de recuperación de errores sintácticos). El punto de partida es la configuración:

$$(s_0 \quad , \quad a_1\ a_2\ \ldots\ a_n\ \$)$$

donde s_0 es el estado inicial del autómata.

Vamos a ilustrar con un ejemplo la aplicación de este método. Para ello partiremos de la gramática del cuadro 3.4. Dicha gramática tiene asociadas dos tablas una de ACCION y otra de GOTO. Por regla general, y por motivos de espacio, ambas tablas suelen fusionarse en una sola. Una tabla tal se divide en dos partes: unas columnas comunes a ACCION y GOTO, y otras columnas sólo de GOTO. Las casillas de las columnas que son sólo de GOTO contienen números de estados a los que se transita; si están vacías quiere decir que hay un error. Las columnas comunes a ambas tablas pueden contener:

① E	→	E + T
②	|	T
③ T	→	T * F
④	|	F
⑤ F	→	(E)
⑥	|	id

Cuadro 3.4 Gramática no ambigua que reconoce expresiones aritméticas en las que sólo intervienen identificadores

- D-i: significa “desplazar y pasar al estado i” (ACCION y GOTO todo en uno, para ahorrar espacio).
- R j: significa “reducir por la regla de producción número j”. En este caso debe

aplicarse a continuación la tabla GOTO.

- Aceptar: la gramática acepta la cadena de terminales y finaliza el proceso de análisis.
- En blanco: rechaza la cadena y finaliza el proceso (no haremos recuperación de errores por ahora).

La tabla asociada a la gramática del cuadro 3.4 siguiendo los criterios anteriormente expuestos es:

	tabla ACCIÓN-GOTO						tabla GOTO		
ESTADO	id	+	*	(	)	$	E	T	F
0	D-5			D-4			1	2	3
1		D-6				Aceptar			
2		R2	D-7		R2	R2			
3		R4	R4		R4	R4			
4	D-5			D-4			8	2	3
5		R6	R6		R6	R6			
6	D-5			D-4				9	3
7	D-5			D-4					10
8		D-6			D-11				
9		R1	D-7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Supongamos ahora que se desea reconocer o rechazar la secuencia “id*(id+ id)”, y que el estado s_0 es, en nuestro caso, el 0. Así, la siguiente tabla muestra un ciclo del algoritmo por cada fila.

Pila	β	ACCION-GOTO
0	id * (id + id)$	D-5
0 id-5	* (id + id) $	R6-3
0 F-3	* (id + id) $	R4-2
0 T-2	* (id + id) $	D-7
0 T-2 *-7	(id + id) $	D-4
0 T-2 *-7 (-4	id + id) $	D-5
0 T-2 *-7 (-4 id-5	+ id) $	R6-3
0 T-2 *-7 (-4 F-3	+ id) $	R4-2
0 T-2 *-7 (-4 T-2	+ id) $	R2-8
0 T-2 *-7 (-4 E-8	+ id) $	D-6
0 T-2 *-7 (-4 E-8 +-6	id) $	D-5
0 T-2 *-7 (-4 E-8 +-6 id-5	) $	R6-3
0 T-2 *-7 (-4 E-8 +-6 F-3	) $	R4-9
0 T-2 *-7 (-4 E-8 +-6 T-9	) $	R1-8
0 T-2 *-7 (-4 E-8	) $	D-11
0 T-2 *-7 (-4 E-8) 11	$	R5-10

0 T-2 *-7 F-10	$	R3-2
0 T-2	$	R2-1
0 E-1	$	Aceptar

Nótese que con este método, la pila hace las funciones de α. La diferencia estriba en la existencia de estados intercalados: hay un estado inicial en la base de la pila, y cada símbolo de α tiene asociado un estado. Nótese, además, que cuando se reduce por una regla, el consecuente de la misma coincide con el extremo derecho de α.

3.5.6.1 Consideraciones sobre el análisis LR(1)

Uno de los primeros pasos que se deben dar durante la construcción de un traductor consiste en la adecuada selección de la gramática que reconozca el lenguaje. Y no es un paso trivial ya que, aunque por regla general existe una multitud de gramáticas equivalentes, cuando se trabaja con aplicaciones prácticas las diferencias de comportamientos entre gramáticas equivalentes adquiere especial relevancia.

En este apartado se dan un par de nociones sobre cómo elegir correctamente las gramáticas.

3.5.6.1.1 Recursión a derecha o a izquierda

Uno de los planteamientos más elementales consiste en decidir si utilizar reglas recursivas a izquierda o recursivas a derecha. Por ejemplo, para reconocer una secuencia de identificadores separados por comas como:

id, id, id

es posible optar por dos gramáticas diferentes:

a) ① lista → ID
② | lista ‘,’ ID

b) ① lista → ID
② | ID ‘,’ lista

Pues bien, a la hora de realizar el reconocimiento de una sentencia podemos observar que se obtienen árboles bien distintos, como en la figura 3.26 al reconocer “id, id, id”. A su vez, si aplicamos un reconocimiento LR, la secuencia de pasos aplicada difiere radicalmente: en el caso b) se desplazan secuencialmente, uno a uno todos los *tokens* de la cadena y, por último, se hacen varias reducciones seguidas, según la longitud de la cadena; en el caso a) se desplazan varios *tokens* (la coma y un ID) y se reduce por ②, y así sucesivamente se van alternando los desplazamientos y las reducciones.

De esta manera, con recursión a la derecha, la pila α va aumentando de tamaño hasta que comienzan las reducciones. Por tanto, ante listas de identificadores lo suficientemente grandes podría producirse un desbordamiento. Lo más conveniente, por tanto, es utilizar gramáticas recursivas a la izquierda para que el tamaño de la pila se mantenga estable. En aquellos casos en que la recursión a la derecha sea absolutamente necesaria (algunos de estos casos se nos plantearán en capítulos

posteriores) debe evitarse el suministro de cadenas excesivamente largas y que puedan producir desbordamientos.

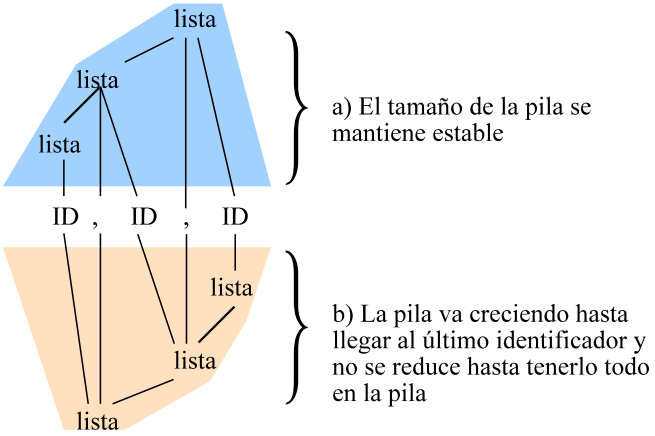


Figura 3.26. Reconocimiento de una misma sentencia por gramáticas equivalentes: a) recursiva a la izquierda, y b) recursiva a la derecha

3.5.6.1.2 Conflictos

Un conflicto se produce cuando el analizador no es capaz de generar la tabla de chequeo de sintaxis para una gramática. Básicamente pueden darse dos tipos de conflictos:

- Desplazar/Reducir (*Shift/Reduce*). Suelen deberse a gramáticas ambiguas.
- Reducir/Reducir (*Reduce/Reduce*). Debidos a que la gramática no admite un análisis LR(k).

El conflicto Desplazar/Reducir aparece cuando en la tabla de acciones aparece en la misma casilla tanto una R de reducir como una D de desplazar, el conflicto es que el programa no sabe si reducir o desplazar. Por ejemplo, una gramática tan sencilla como:

EXPR → EXPR '+' EXPR

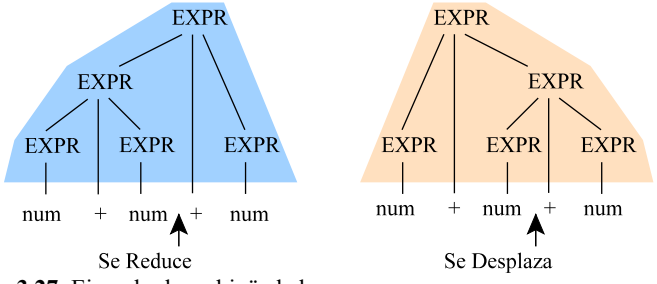


Figura 3.27. Ejemplo de ambigüedad

| num

resulta obviamente ambigua (es recursiva por la derecha y por la izquierda simultáneamente), luego ante una entrada tan sencilla como "num + num + num" pueden producirse dos árboles sintácticos diferentes (ver figura 3.27).

Por otro lado, el conflicto Reducir/Reducir aparece cuando el analizador puede aplicar dos o más reglas ante determinadas situaciones. Esta situación suele aparecer cuando la gramática no es LR(1), esto es, no es suficiente con un *token* de pre-búsqueda para tomar la decisión de qué acción realizar. Por ejemplo, supongamos la gramática:

① S → aaBdd
② | aCd
③ B → a
④ C → aa

que únicamente reconoce las dos secuencia de *tokens* "aaadd" y "aaad". Pues bien, una gramática tan sencilla no es LR(1) (ver figura 3.28). Supongamos que en esta figura se desea reconocer la secuencia "aaad"; en tal caso lo correcto sería reducir por ④. Sin embargo todo depende de si detrás del *token* de pre-búsqueda (una "d"), viene otra "d" o lleva EOF. Un analizador LR(2) sería capaz de tomar la decisión correcta, pero no uno LR(1).

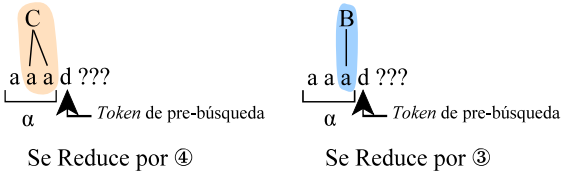


Figura 3.28. Ejemplo de conflicto Reduce/Reduce. Las interrogaciones indican que el analizador desconoce lo que viene a continuación

Los conflictos Reduce/Reduce se pueden eliminar en la mayoría de los casos. Esto se consigue retrasando la decisión de reducción al último momento. P.ej., la gramática:

S → aaadd
| aaad

es equivalente a la anterior, pero sí es LR(1), ya que sólo se decide reducir ante el EOF del final, en cuyo momento ya se estará en el estado correcto en función de los *tokens* que se han leído al completo.

3.5.6.2 Conclusiones sobre el análisis LR(1)

En el epígrafe 3.5.6 se estudió cómo funcionan los analizadores LR mediante la utilización de sus correspondientes tablas de análisis. En el ejemplo final puede observarse como el estado que siempre se encuentra en cabeza de la pila contiene en

todo momento la información necesaria para la reducción, si esto procede. Además, hemos dejado al margen intencionadamente la estructura del programa de proceso puesto que se trata esencialmente de un autómata finito con pila.

En general puede afirmarse que, dada la estructura de los analizadores LR, con la sola inspección de k símbolos de la cadena de entrada a la derecha del último consumido puede decidirse con toda exactitud cual es el movimiento a realizar (reducción, desplazamiento, etc). Es por este motivo por lo que suele denominarse a este tipo de gramáticas como LR(k). Como ya se comentó, en la práctica casi todos los lenguajes de programación pueden ser analizados mediante gramáticas LR(1).

Las tablas LR(1) ideadas por Knuth en 1965 son demasiado grandes para las gramáticas de los lenguajes de programación tradicionales. En 1969 De Remer y Korenjack descubrieron formas de compactar estas tablas, haciendo práctico y manejable este tipo de analizadores. El algoritmo que se aplica sobre dichas tablas es el mismo ya visto. También hemos visto que hay tres tipos principales de gramáticas LR(k) que varían en función de su potencia estando incluidas unas en otras de la siguiente forma:

$$SLR(k) \subset LALR(k) \subset LR(k)$$

pero no sus lenguajes respectivos que coinciden en sus conjuntos, esto es, si hay una gramática LR(1) que reconoce el lenguaje L, entonces también hay una gramática SLR(1) que reconoce a L.

Los metacompiladores Yacc y Cup utilizan el análisis LALR(1). El autómata debe de mantener información de su configuración (pila α), y para mantener información sobre β se comunica con Lex, quien se encarga de la metacompilación a nivel léxico.

Capítulo 4

Gramáticas atribuidas

4.1 Análisis semántico

Además de controlar que un programa cumple con las reglas de la gramática del lenguaje, hay que comprobar que lo que se quiere hacer tiene sentido. El análisis semántico dota de un significado coherente a lo que hemos hecho en el análisis sintáctico. El chequeo semántico se encarga de que los tipos que intervienen en las expresiones sean compatibles o que los parámetros reales de una función sean coherentes con los parámetros formales: p.ej. no suele tener mucho sentido el multiplicar una cadena de caracteres por un entero.

Comenzaremos viendo un ejemplo sencillo en el que se introduce el concepto de atributo mediante la construcción del intérprete de una calculadora.

4.1.1 Atributos y acciones semánticas

A continuación se desarrolla un ejemplo para tener una visión global de este tema. Supongamos que se desea construir una pequeña calculadora que realiza las operaciones + y * . La gramática será la siguiente:

```
E  →  E + T
   |  T
T  →  T * F
   |  F
F  →  ( E )
   |  num
```

y queremos calcular: 33 + 12 + 20. Los pasos a seguir son:

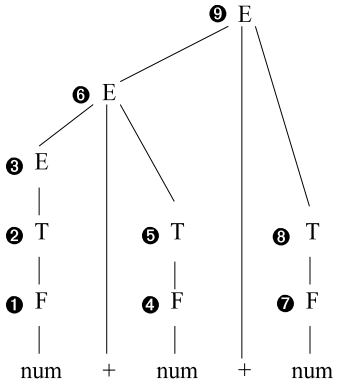


Figura 4.1. Árbol de reconocimiento de "num + num + num". Los números entre círculos indican la secuencia de reducciones según un análisis LR(1)

1. Análisis léxico. La cadena de entrada se convierte en una secuencia de *tokens*, p.ej. con PCLex: 33 + 12 + 20 ➪ num + num + num

2. Análisis sintáctico. Mediante la gramática anterior y, p. ej., PCYacc se produce el árbol sintáctico de la figura 4.1.

Hemos conseguido construir el árbol sintáctico, lo que no quiere decir que sea semánticamente correcto. Si deseamos construir un intérprete, el siguiente paso es saber qué resultado nos da cada operación, y para ello hemos etiquetado el árbol indicando el orden en el que se han aplicado las reducciones. El orden en el que se van aplicando las reglas nos da el *parse* derecho.

El orden en que se aplican las reglas de producción está muy cercano a la semántica de lo que se va a reconocer. Y este hecho debemos aprovecharlo. Para ello vamos a decorar el árbol sintáctico asociando valores a los terminales y no terminales que en él aparecen. A los terminales le asignaremos ese valor mediante una acción asociada al patrón correspondiente en el análisis léxico, de la siguiente forma:

```
[0-9]+    {
          Convertir yytext en un entero y asociar dicho entero al token NUM;
          return NUM;
          }
```

por lo tanto lo que le llegará al analizador sintáctico es:

$$\text{num}._{33} +_{\cdot\text{nada}} \text{num}._{12} +_{\cdot\text{nada}} \text{num}._{20}$$

Vamos a ver qué uso podría hacerse de éstos **atributos** para, a medida que se hace el reconocimiento sintáctico, ir construyendo el resultado a devolver. La

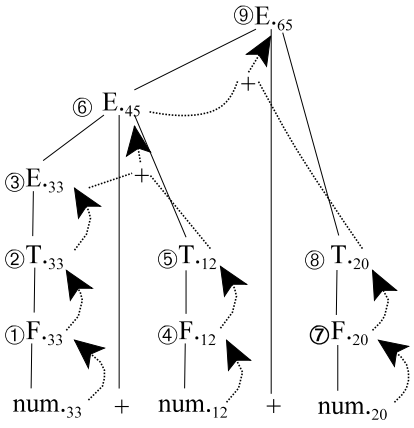


Figura 4.2. Propagación de atributos de las hojas hacia la raíz

idea es que cada no terminal de la gramática tenga asociado un atributo que aglutine, de alguna manera, la información del sub-árbol sintáctico del que es raiz.

La figura 4.2 ilustra la propagación de atributos desde los terminales hacia los no terminales. Esta propagación puede implementarse ejecutando acciones asociadas a cada regla de producción, de tal manera que la acción se ejecutará cada vez que se reduzca por su regla asociada, de igual manera que podíamos asociar una acción a un patrón del analizador léxico. De esta manera, con las asociaciones de valores a los terminales y no terminales y las asociaciones de **acciones semánticas** a los patrones y a las reglas de producción, podemos evaluar el comportamiento de un programa, es decir, generamos código (concreto o abstracto; podemos considerar que un intérprete genera código abstracto, ya que en ningún momento llega a existir físicamente). Esto puede verse en la siguiente tabla (los subíndices tan sólo sirven para diferenciar entre distintas instancias de un mismo símbolo gramatical cuando aparece varias veces en la misma regla).

Reglas de producción	Acciones semánticas asociadas
$E_1 \rightarrow E_2 + T$	$\{E_1 = E_2 + T;\}$
$E \rightarrow T$	$\{E = T;\}$
$T_1 \rightarrow T_2 * F$	$\{T_1 = T_2 * F;\}$
$T \rightarrow F$	$\{T = F;\}$
$F \rightarrow (E)$	$\{F = E;\}$
$F \rightarrow num$	$\{F = NUM;\}$

Estas acciones se deben ejecutar cada vez que se reduce sintácticamente por la regla asociada.

Basándonos en el funcionamiento de un analizador con funciones recursivas, es como si cada función retornase un valor que representa al sub-árbol del que es raíz. La implementación en el caso LR(1) es un poco más compleja, almacenándose los atributos en la pila α de manera que, excepto por el estado inicial, α almacena tuplas de la forma (estado, símbolo, atributo), con lo que todo símbolo tiene un atributo asociado incluso cuando éste no es necesario para nada (como en el caso del *token* + del ejemplo anterior). Los cambios producidos se ilustran en la figura 4.3.

Así, un **atributo** es una información asociada a un terminal o a un no terminal. Una **acción** o **regla semántica** es un algoritmo que puede acceder a los atributos de los terminales y/o no terminales. Como acción semántica no sólo se puede poner una asignación a un atributo, además puede añadirse código; p.ej. si se hace la asociación:

① $E_1 \rightarrow E_2 + T$ $\{E_1 = E_2 + T;$ printf("%d\n", $E_1)\}$

utilizando notación pseudo-C, entonces cada vez que se aplique la regla ① se visualizará en pantalla el valor del atributo de la regla que, en el ejemplo, serán los valores 45 y 65. Si sólo se deseara que apareciera el valor resultante global, podría añadirse una nueva regla S → E y pasar el **printf** de la regla ① a esta otra:

S → E { printf("%d\n", E); }

$E_1 \rightarrow E_2 + T \qquad \{E_1 = E_2 + T; \}$

De esta forma, la construcción del árbol sintáctico tendría una reducción más, y cuando ésta se hiciera aparecería el resultado por pantalla. Nótese como la adición de esta regla no modifica la gramática en el sentido de que sigue reconociendo el mismo lenguaje pero, sin embargo, nos proporciona una clara utilidad al permitir visualizar únicamente el resultado final del cálculo.

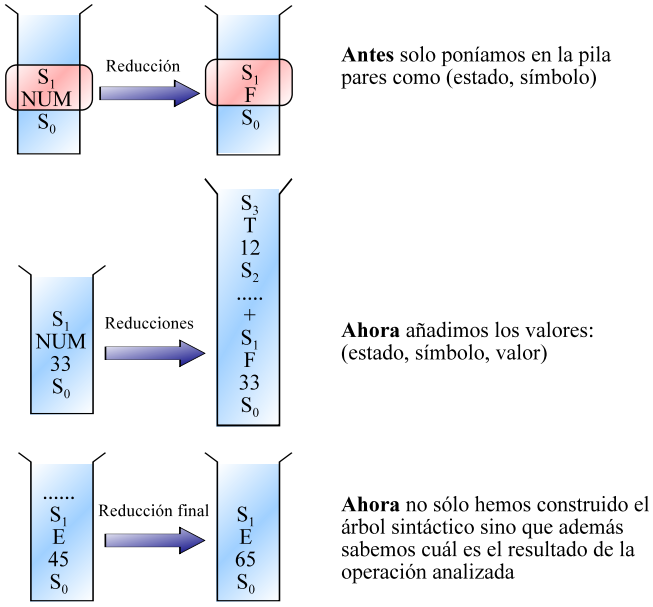


Figura 4.3. Gestión de la pila α sin atributos y con atributos

4.1.2 Ejecución de una acción semántica

Hay dos formas de asociar reglas semánticas con reglas de producción:

- **Definición dirigida por sintaxis**: consiste en asociar una acción semántica a una regla de producción, pero dicha asociación no indica cuándo se debe ejecutar dicha acción semántica. Se supone que en una primera fase se construye el árbol sintáctico completo y, posteriormente, se ejecutan las acciones semánticas en una secuencia tal que permita el cálculo de todos los atributos de los nodos del árbol.
- **Esquema de traducción**: es igual que una definición dirigida por sintaxis excepto que, además, se asume o se suministra información acerca de cuándo se deben ejecutar las acciones semánticas. Es más, también es posible intercalar acciones entre los símbolos del consecuente de una regla de

producción, ej.: A → c d {A = c + d} B. En el caso LR las acciones se ejecutan cuando se efectúan las reducciones. En el caso con funciones recursivas las acciones se encuentran intercaladas con el código destinado a reconocer cada expresión BNF.

Siempre que tengamos un esquema de traducción con acciones intercaladas se puede pasar a su equivalente definición dirigida por sintaxis. Por ejemplo, el esquema:

$A \rightarrow c\ d\ \{\ acción_1\ \}\ B$

se puede convertir en

$A \rightarrow \quad c\ d\ N_1\ B$

$N_1 \rightarrow \quad \varepsilon\ \{\ acción_1\ \}$

que es equivalente. Básicamente, la transformación consiste en introducir un no terminal ficticio (N_1 en el ejemplo) que sustituye a cada acción semántica intermedia ($acción_1$), y crear una nueva regla de producción de tipo épsilon con dicho no terminal como antecedente y con dicha acción semántica como acción.

A primera vista esta transformación puede parecer que puede dar problemas; p.ej. al convertir el esquema:

$A \rightarrow c\ d\ \{$ printf("%d\n", c+d); $\}\ B$

en:

$A \rightarrow c\ d\ N_1\ B$

$N_1 \rightarrow \varepsilon\ \{$ printf("%d\n", c+d); $\}$

aparece una acción semántica que hace referencia a los atributos de c y d asociada a una regla en la que c y d no intervienen. Sin embargo, dado que sólo existe una regla en la que N_1 aparece en el antecedente y también sólo una regla en la que N_1 aparece en el consecuente, está claro que siempre que un analizador intente reducir a N_1 o invocar a la función recursiva que implementa a N_1, los dos últimos *tokens* que hayan sido consumidos serán c y d, por lo que la acción semántica puede ejecutarse trabajando sobre los atributos de estos dos últimos *tokens*, aunque aparentemente parezca que hay algún problema cuando se echa un vistazo a la ligera a la definición dirigida por sintaxis obtenida tras la transformación del esquema de traducción.

Cuando se trabaja con reglas de producción (no con funciones recursivas), podemos encontrarnos con algún otro problema. P.ej., si se tiene:

$A \rightarrow c\ d\ \{A = c + d\}\ B$

y se transforma a:

$A \rightarrow c\ d\ N_1\ B$

$N_1 \rightarrow \varepsilon\ \{A = c + d\}$

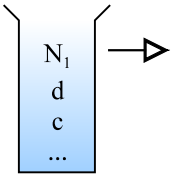


Figura 4.4. Esquema de la pila α tras reducir por la regla $N_1 \rightarrow \varepsilon$. La acción semántica no puede acceder al atributo de A porque A no ha sido reducida todavía y no está en la pila

entonces en la pila se tiene lo que ilustra la figura 4.4.

En general, la utilización de atributos en un esquema de traducción obedece a reglas del sentido común una vez que se conocen las transformaciones que un metacompilador aplica automáticamente a las acciones semántica intermedias, ya se trate de analizadores LR o con funciones recursivas. En el caso del análisis ascendente la cosa puede no estar tan clara de primera hora y resulta conveniente exponer claramente cuáles son estas **reglas sobre la utilización de atributos en acciones semánticas de un esquema LR**:

- Un atributo solo puede ser usado (en una acción) detrás del símbolo al que pertenece.
- El atributo del antecedente solo se puede utilizar en la acción (del final) de su regla. Ej.:

  ✔ A → c d N_1 B { sólo aquí se puede usar el atributo de A }

  ✘ N1 → ε { aquí no se puede usar el atributo de A }
- En una acción intermedia sólo se puede hacer uso de los atributos de los simbolos que la preceden. Siguiendo los ejemplo anteriores, si se hace una sustitución de una acción intermedia y se obtiene:

  A → c d N_1

  N_1 → ε

  pues en la acción de N_1 sólo se puede hacer uso de los atributos de c y d (que es lo que le precede).

  Todo esto se verá en epígrafes posteriores con mayor nivel de detalle.

4.2 Traducción dirigida por sintaxis

A partir de este epígrafe se desarrolla la traducción de lenguajes guiada por gramáticas de contexto libre. Se asocia información a cada construcción del lenguaje de programación proporcionando atributos a los símbolos de la gramática que representan dicha construcción. Los valores de los atributos se calculan mediante reglas (acciones) semánticas asociadas a las reglas de producción gramatical.

Como hemos visto, hay dos notaciones para asociar reglas semánticas con reglas de producción:

- Las definiciones dirigidas por sintaxis son especificaciones de alto nivel para traducciones. No es necesario que el usuario especifique explícitamente el orden en el que tiene lugar la ejecución de las acciones.
- Los esquemas de traducción indican el orden en que se deben evaluar las reglas semánticas.

Conceptualmente, tanto con las definiciones dirigidas por sintaxis como con los esquemas de traducción, se analiza sintácticamente la cadena de componentes léxicos de entrada, se construye el árbol de análisis sintáctico y después se recorre el árbol para evaluar las reglas semánticas en sus nodos. La evaluación de las reglas

semánticas puede generar código, guardar información en una tabla de símbolos, emitir mensajes de error o realizar otras actividades, como p.ej. evaluar expresiones aritméticas en una calculadora. La traducción de la cadena de componentes léxicos es el resultado obtenido al evaluar las reglas semánticas.

En la práctica, no todas las implementaciones tienen que seguir al pie de la letra este esquema tan nítido de una fase detrás de otra. Hay casos especiales de definiciones dirigidas por la sintaxis que se pueden implementar en una sola pasada evaluando las reglas semánticas durante el análisis sintáctico, sin construir explícitamente un árbol de análisis sintáctico o un grafo que muestre las dependencias entre los atributos.

4.2.1 Definición dirigida por sintaxis

Una definición dirigida por sintaxis es una gramática de contexto libre en la que cada símbolo gramatical (terminales y no terminales) tiene un conjunto de atributos asociados, dividido en dos subconjuntos llamados **atributos sintetizados** y **atributos heredados** de dicho símbolo gramatical. Si se considera que cada nodo de un árbol sintáctico tiene asociado un registro con campos para guardar información, entonces un atributo corresponde al nombre de un campo.

Un atributo puede almacenar cualquier cosa: una cadena, un número, un tipo, una posición de memoria, etc. El valor de un atributo en un nodo de un árbol sintáctico se define mediante una regla semántica asociada a la regla de producción utilizada para obtener dicho nodo. El valor de un **atributo sintetizado** de un nodo se calcula a partir de los valores de los atributos de sus nodos hijos en el árbol sintáctico; el valor de un **atributo heredado** se calcula a partir de los valores de los atributos en los nodos hermanos y padre. Cada nodo del árbol representa una instancia de un símbolo gramatical y, por tanto, tiene sus propios valores para cada atributo.

Las **reglas semánticas** establecen las dependencias entre los atributos; dependencias que pueden representarse mediante un grafo, una vez construido el árbol sintáctico al completo. Del grafo de dependencias se obtiene un orden de evaluación de todas las reglas semánticas. La evaluación de las reglas semánticas define los valores de los atributos en los nodos del árbol de análisis sintáctico para la cadena de entrada. Una regla semántica también puede tener efectos colaterales, como por ejemplo visualizar un valor o actualizar una variable global. Como se indicó anteriormente, en la práctica, un traductor no necesita construir explícitamente un árbol de análisis sintáctico o un grafo de dependencias; sólo tiene que producir el mismo resultado para cada cadena de entrada.

Un árbol sintáctico que muestre los valores de los atributos en cada nodo se denomina un **árbol sintáctico decorado** o con anotaciones. El proceso de calcular los valores de los atributos en los nodos se denomina anotar o **decorar el árbol sintáctico**.

4.2.2 Esquema formal de una definición dirigida por sintaxis

Formalmente, en una definición dirigida por sintaxis, cada regla de producción

$A \rightarrow \alpha$

tiene asociado un conjunto de reglas semánticas siendo cada una de ellas de la forma

$b := \mathbf{f}(c_1, c_2, ..., c_k)$

donde **f** es una función y, o bien:

1.- b es un **atributo sintetizado** de A y $c_1, c_2, ..., c_k$ son atributos de los símbolos de α, o bien

2.- *b* es un **atributo heredado** de uno de los símbolos de α , y $c_1, c_2, ..., c_k$ son atributos de los restantes símbolos de α o bien de A.

En cualquier caso, se dice que el atributo b depende de los atributos $c_1, c_2, ..., c_k$.

Una **gramática con atributos** es una definición dirigida por sintaxis en la que las funciones en las reglas semánticas no pueden tener efectos colaterales.

Las asignaciones de las reglas semánticas a menudo se escriben como expresiones. Ocasionalmente el único propósito de una regla semántica en una definición dirigida por sintaxis es crear un efecto colateral. Dichas reglas semánticas se escriben como llamadas a procedimientos o fragmentos de programa. Se pueden considerar como reglas que definen los valores de atributos sintetizados ficticios del no terminal del antecedente de la regla de producción asociada, de manera que no se muestran ni el atributo ficticio ni el signo := de la regla semántica.

P.ej., la definición dirigida por sintaxis del cuadro 4.1 es para un traductor que implementa una calculadora. Esta definición asocia un atributo sintetizado val con valor entero a cada uno de los no terminales E, T y F. Para cada regla de producción de E, T y F, la regla semántica calcula el valor del atributo val para el antecedente a partir de los valores de los atributos del consecuente.

El componente léxico num tiene un atributo sintetizado val_lex cuyo valor viene proporcionado por el analizador léxico. La regla asociada a la producción

$L \rightarrow E$ '\n'

donde L es el axioma inicial, es sólo un procedimiento que visualiza como resultado el valor de la expresión aritmética generada por E; se puede considerar que esta regla define un falso atributo para el no terminal L. Más adelante veremos una especificación en PCYacc para esta calculadora, y ello nos llevará a ilustrar la traducción durante el

Reglas de producción	Acciones semánticas asociadas
$L \rightarrow E$ '\n'	printf("%d\n", E.val)
$E_1 \rightarrow E_2 + T$	E_1.val = E_2.val + T.val
$E \rightarrow T$	E.val = T.val
$T_1 \rightarrow T_2 * F$	T_1.val = T_2.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ num	F.val = num.val_lex

Cuadro 4.1 Definición dirigida por sintaxis de una calculadora sencilla

análisis sintáctico LALR(1). También veremos un ejemplo con JavaCC pero, por ahora, seguiremos profundizando en las definiciones dirigidas por sintaxis.

En una definición dirigida por sintaxis, se asume que los *tokens* sólo tienen atributos sintetizados, ya que la definición no proporciona ninguna regla semántica para ellos. El analizador léxico es el que proporciona generalmente los valores de los atributos de los terminales. Además se asume que el símbolo inicial no tiene ningún atributo heredado, a menos que se indique lo contrario, ya que en muchos casos suele carecer de padre y de hermanos.

4.2.2.1 Atributos sintetizados

Los atributos sintetizados son muy utilizados en la práctica. Una definición dirigida por sintaxis que usa atributos sintetizados exclusivamente se denomina **definición con atributos sintetizados**. Con una definición con atributos sintetizados se puede decorar un árbol sintáctico mediante la evaluación de las reglas semánticas de forma ascendente, calculando los atributos desde los nodos hoja a la raíz.

Por ejemplo, la definición con atributos sintetizados del cuadro 4.1 especifica una calculadora que lee una línea de entrada que contiene una expresión aritmética que incluye dígitos (asumimos que el *token* num está asociado al patrón [0-9]+), paréntesis, los operadores + y *, seguida de un carácter de retorno de carro('\n'), e imprime el valor de la expresión. Por ejemplo, dada la cadena "3*5+4" seguida de un retorno de carro, el sistema visualiza el valor 19. La figura 4.5 contiene el árbol sintáctico decorado para la entrada propuesta. El resultado, que se imprime en la raíz del árbol, es el valor de E.val.

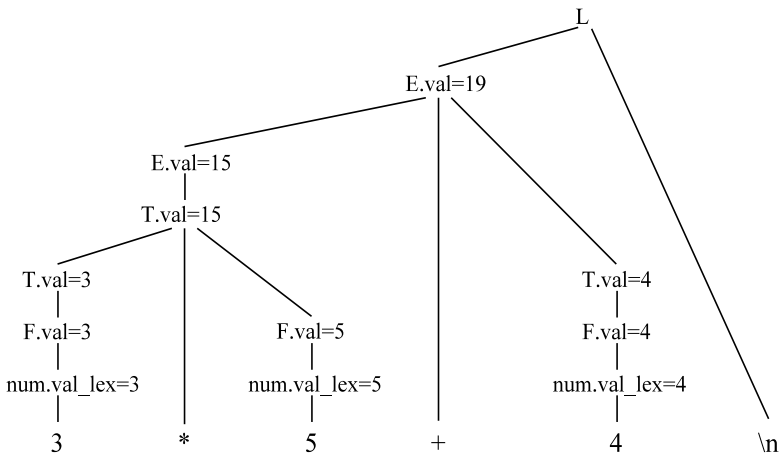


Figura 4.5. Árbol de análisis sintáctico con anotaciones para reconocer "3*5+4\n".

Aunque una definición dirigida por sintaxis no dice nada acerca de cómo ir

calculando los valores de los atributos, está claro que debe existir una secuencia viable de cálculos que permita realizar dichos cálculos. Vamos a ver una posible secuencia. Considérese el nodo situado en el extremo inferior izquierdo que corresponde al uso de la producción F → num. La regla semántica correspondiente, F.val:= num.val_lex, establece que el atributo F.val en el nodo tenga el valor 3 puesto que el valor de num.val_lex en el hijo de este nodo es 3. De forma similar, en el padre de este F, el atributo T.val tiene el valor 3.

A continuación considérese el nodo raíz de la producción T → T * F. El valor del atributo T.val en este nodo está definido por T.val := T_1.val * F.val. Cuando se aplica la regla semántica en este nodo, T_1.val tiene el valor 3 y F.val el valor 5. Por tanto, T.val adquiere el valor 15 en este nodo.

Por último, la regla asociada con la producción para el no terminal inicial, L → E \n, visualiza el valor de la expresión representada por E, esto es E.val.

4.2.2.2 Atributos heredados

Un atributo heredado es aquél cuyo valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos de su padre y/o de sus hermanos. Los atributos heredados sirven para expresar la dependencia de una construcción de un lenguaje de programación en el contexto en el que aparece. Por ejemplo, se puede utilizar un atributo heredado para comprobar si un identificador aparece en el lado izquierdo o en el derecho de una asignación para decidir si se necesita la dirección o el valor del identificador. Aunque siempre es posible reescribir una definición dirigida por sintaxis para que sólo se utilicen atributos sintetizados, a veces es más natural utilizar definiciones dirigidas por la sintaxis con atributos heredados.

En el siguiente ejemplo, un atributo heredado distribuye la información sobre los tipos a los distintos identificadores de una declaración. El cuadro 4.2 muestra una gramática que permite una declaración, generada por el axioma D, formada por la palabra clave int o real seguida de una lista de identificadores separados por comas.

Reglas de producción	Acciones semánticas asociadas
D → T L	L.her:= T.tipo
T → int	T.tipo := “integer”
T → real	T.tipo := “real”
L_2 → L_1 , id	L_1.her := L_2.her añadetipo(id.ptr_tds, L_2.her)
L → id	añadetipo(id.ptr_tds, L.her)

Cuadro 4.2 Definición dirigida por sintaxis con el atributo heredado L.her

El no terminal T tiene un atributo sintetizado tipo, cuyo valor viene determinado por la regla por la que se genera. La regla semántica L.her := T.tipo, asociada con la regla de producción D → T L, asigna al atributo heredado L.her el tipo

de la declaración. Entonces las reglas pasan hacia abajo este valor por el árbol sintáctico utilizando el atributo heredado L.her. Las reglas asociadas con las producciones de L llaman al procedimiento añadetipo, que suponemos que añade el tipo de cada identificador a su entrada en la tabla de símbolos (apuntada por el atributo ptr_tds). Este último detalle carece de importancia para el asunto que estamos tratando en este momento y se verá con mayor detenimiento en capítulos posteriores.

En la figura 4.6 se muestra un árbol de análisis sintáctico con anotaciones para la cadena "real id_1, id_2, id_3". El valor de L.her en los tres nodos de L proporciona el tipo a los identificadores id_1, id_2 e id_3. Estos valores se obtienen calculando el valor del atributo T.tipo en el hijo izquierdo de la raíz y evaluando después L.her de forma descendente en los tres nodos de L en el subárbol derecho de la figura. En cada nodo de L también se llama al procedimiento añadetipo para indicar en la tabla de símbolos el hecho de que el identificador en el hijo de cada nodo L tiene tipo real.

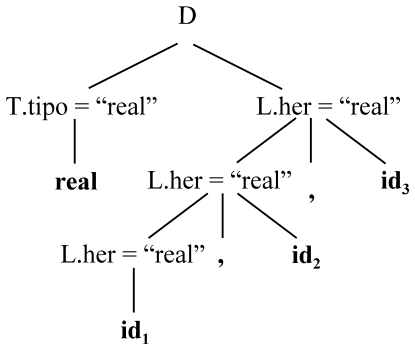


Figura 4.6. Árbol sintáctico con el atributo heredado L.her en cada nodo L

4.2.2.3 Grafo de dependencias

Si un atributo *b* en un nodo de un árbol sintáctico depende de un atributo *c*, entonces se debe evaluar la regla semántica para *b* en ese nodo después de la regla semántica que define a *c*. Las interdependencias entre los atributos en los nodos de un árbol sintáctico pueden representarse mediante un grafo dirigido llamado **grafo de dependencias**.

Antes de construir el grafo de dependencias para un árbol sintáctico, se escribe cada regla semántica en la forma b:=f(c_1, c_2, ..., c_k), introduciendo un falso atributo sintetizado *b* para cada regla semántica que conste de una llamada a procedimiento. El grafo tiene un nodo por cada atributo de un nodo del árbol y una arista desde el nodo *c* al *b* si el atributo *b* depende del atributo *c*. Algorítmicamente, el grafo de dependencias para un determinado árbol de análisis sintáctico se construye de la siguiente manera:

Para cada nodo *n* en el árbol de análisis sintáctico
    Para cada atributo *a* del símbolo gramatical en el nodo *n*

construir un nodo en el grafo de dependencias para *a*;
Para cada nodo *n* en el árbol de análisis sintáctico
Para cada regla semántica $b:=f(c_1, c_2, ..., c_k)$ asociada con la producción utilizada en *n*
Para *i*:= 1 **hasta** *k*
colocar un arco desde el nodo c_i hasta el nodo *b*

Por ejemplo, supóngase que A.a := f(X.x, Y.y) es una regla semántica para la producción A → X Y. Esta regla define un atributo sintetizado *A.a* que depende de los atributos *X.x* e *Y.y*. Si se utiliza esta producción en el árbol de análisis sintáctico, entonces habrá tres nodos, *A.a*, *X.x* e *Y.y*, en el grafo de dependencias con un arco desde *X.x* hasta *A.a* puesto que *A.a* depende de *X.x* y otro arco desde *Y.y* hasta *A.a* puesto que *A.a* también depende de *Y.y*.

Si la regla de producción A→ XY tuviera asociada la regla semántica X.i := g(A.a, Y.y), entonces habría un arco desde *A.a* hasta *X.i* y otro arco desde *Y.y* hasta *X.i*, puesto que *X.i* depende tanto de *A.a* como de *Y.y*.

Los arcos que se añaden a un grafo de dependencias son siempre los mismos para cada regla semántica, o sea, siempre que se utilice la regla de producción del

Regla de producción	Acción semántica asociada
$E_3 \rightarrow E_1 + E_2$	$E_3.val := E_1.val + E_2.val$

Cuadro 4.3 Ejemplo de regla de producción con acción semántica asociada

cuadro 4.3 en un árbol sintáctico, se añadirán al grafo de dependencias los arcos que se muestran en la figura 4.7.

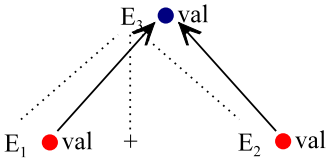


Figura 4.7. $E_3.val$ se sintetiza a partir de $E_1.val$ y $E_2.val$

Los tres nodos del grafo de dependencias (marcados con un círculo de color) representan los atributos sintetizados $E_3.val$, $E_1.val$ y $E_2.val$ en los nodos correspondientes del árbol sintáctico. El arco desde $E_1.val$ hacia $E_3.val$ muestra que $E_3.val$ depende de $E_1.val$ y el arco hacia $E_3.val$ desde $E_2.val$ muestra que $E_3.val$ también depende de $E_2.val$. Las líneas de puntos representan al árbol de análisis sintáctico y no son parte del grafo de dependencias.

En la figura 4.8 se muestra el grafo de dependencias para el árbol de análisis sintáctico de la figura 4.6. Los arcos de este grafo están numerados. Hay una arco(②) hacia el nodo L.her desde el nodo T.tipo porque el atributo heredado L.her depende del atributo T.tipo según la regla semántica L.her:=T.tipo de la regla de producción D →

T L. Los dos arcos que apuntan hacia abajo (④ y ⑥) surgen porque L_{i+1}.her depende de L_i.her según la regla semántica L_1.her := L_2.her de la regla de producción $L_2 \rightarrow L_1$, ID. Cada una de las reglas semánticas añadetipo(id.ptr_tds, L.her) asociada con las producciones de L conduce a la creación de un falso atributo. Los nodos amarillos se construyen para dichos falsos atributos (arcos ③, ⑤ y ⑦).

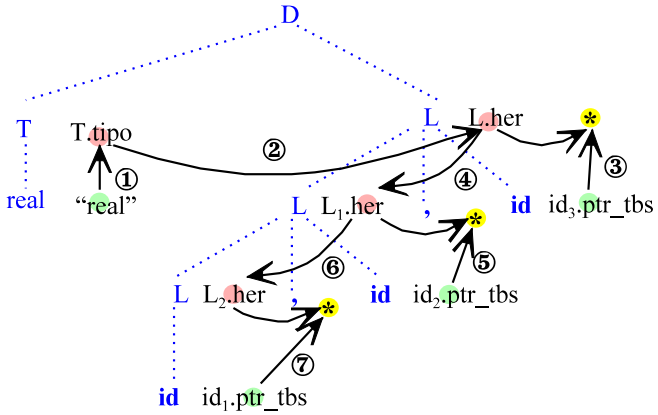


Figura 4.8. Grafo de dependencias para el árbol sintáctico de la figura 4.6. El árbol sintáctico aparece de azul, los atributos de los *tokens* de verde, los atributos ficticios de amarillo, y el resto de atributos se muestra en rosa.

4.2.2.4 Orden de evaluación

Una ordenación topológica de un grafo dirigido acíclico es toda secuencia m_1, m_2, ..., m_k de los nodos del grafo tal que los arcos van desde los nodos que aparecen primero en la ordenación a los que aparecen más tarde; es decir, si $m_i \rightarrow m_j$ es un arco desde m_i a m_j, entonces m_i aparece antes que m_j en la ordenación.

Toda ordenación topológica de un grafo de dependencias da un orden válido en el que se pueden evaluar las reglas semánticas asociadas con los nodos de un árbol sintáctico. Es decir, en el ordenamiento topológico, los atributos dependientes c_1, c_2,..., c_k en una regla semántica b:=f(c_1, c_2,..., c_k) están disponibles en un nodo antes de que se evalúe f.

La traducción especificada por una definición dirigida por sintaxis se puede precisar como sigue:

1.- Se utiliza la gramática subyacente para construir un árbol sintáctico para la entrada.

2.- El grafo de dependencias se construye como se indica más arriba.

3.- A partir de una ordenación topológica del grafo de dependencias, se obtiene un orden de evaluación para las reglas semánticas. La evaluación de las reglas semánticas en este orden produce la traducción de la cadena de

entrada y/o desencadena su interpretación.

Por ejemplo, la numeración de arcos que se hizo en el grafo de dependencias de la figura 4.8, nos suministra una ordenación topológica lo que produce la secuencia:

① T.tipo = "real"
② L.her = T.tipo
③ añadetipo(ID_3.ptr_tbs, L.her);
④ L_1.her = L.her
⑤ añadetipo(ID_2.ptr_tbs, L.her);
⑥ L_2.her = L_1.her
⑦ añadetipo(ID_1.ptr_tbs, L.her);

La evaluación de estas reglas semánticas almacena finalmente el tipo "real" en la entrada de la tabla de símbolos para cada identificador.

Por último, se dice que una gramática atribuida es **circular** si el grafo de dependencias para algún árbol sintáctico generado por la gramática tiene un ciclo. Por ejemplo, si una regla como:

A→ B C D

tiene las reglas semánticas:

A.a = f (B.b, C.c, D.d)
C.c = g (A.a)

entonces cualquier aplicación de la regla dará lugar a un trozo de árbol sintáctico y grafo de dependencias como el de la figura 4.9.

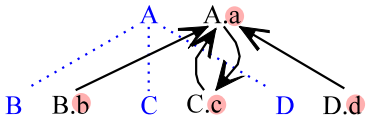


Figura 4.9. Grafo de dependencias circular

Resulta evidente que si el grafo de dependencias tiene un ciclo entonces no es posible encontrar un orden de evaluación. En cualquier otra situación existe, al menos, una ordenación topológica.

4.2.2.5 Gramática L-atribuida

También llamada gramática atribuida por la izquierda, o definición con atributos por la izquierda, es aquélla en la que en toda regla $A \rightarrow X_1 X_2 ... X_n$, cada atributo heredado de X_j, con $1 \leq j \leq n$, depende sólo de:

1.- Los atributos (sintetizados o heredados) de los símbolos $X_1 X_2 ... X_{j-1}$.
2.- Los atributos heredados de A.

Este tipo de gramáticas no produce ciclos y además admiten un orden de evaluación en profundidad mediante el siguiente algoritmo:

Función visitaEnProfundidad(nodo n){
    Para cada hijo m de n, de izquierda a derecha
        evaluar los atributos heredados de m

```
            visitaEnProfundidad(m)
        Fin Para
        evaluar los atributos sintetizados de n
Fin visitaEnProfundidad
```

A continuación se muestra una gramática L-atribuida donde D hace referencia a "declaración", T a "tipo" y L a "lista de identificadores".

$D \rightarrow T\ L$	{L.tipo = T.tipo;}
$T \rightarrow$ integer	{T.tipo = "integer";}
$T \rightarrow$ real	{T.tipo = "real";}
$L \rightarrow L_1$, id	{L_1.tipo = L.tipo; id.tipo = L.tipo;}
$L \rightarrow$ id	{id.tipo = L.tipo;}

Sin embargo, la siguiente gramática no es L-atribuida ya que propaga los atributos de derecha a izquierda:

$D \rightarrow$ id T	{id.tipo = T.tipo; D.tipo = T.tipo;}
$D \rightarrow$ id D_1	{id.tipo = D_1.tipo; D.tipo = D_1.tipo;}
$T \rightarrow$ integer	{T.tipo = "integer";}
$T \rightarrow$ real	{T.tipo = "real";}

4.2.2.6 Gramática S-atribuida

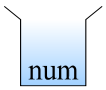
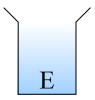
Es una gramática atribuida que sólo contiene atributos sintetizados.Una gramática S-atribuida es también L-atribuida. Los atributos sintetizados se pueden evaluar con un analizador sintáctico ascendente conforme se va construyendo el árbol sintáctico. El analizador sintáctico puede conservar en su pila los valores de los atributos sintetizados asociados con los símbolos gramaticales. Siempre que se haga una reducción se calculan los valores de los nuevos atributos sintetizados a partir de los atributos que aparecen en la pila para los símbolos gramaticales del lado derecho de la regla de producción con la que se reduce.

El cuadro 4.4 muestra un ejemplo de gramática S-atribuida, y la tabla siguiente un ejemplo con el que se pone de manifiesto cómo se propagan los atributos,

Regla de producción	Acción semántica asociada
L → E '\n'	print(E.val)
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val = T_1.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
F → num	F.val = num.val_lex

Cuadro 4.4 Ejemplo de gramática S-atribuida

suponiendo la entrada “3 + 7 \n”.

Árbol	Pila α	Acción y atributos
num | 3		num.val_lex = 3
F | num | 3		F.val = num.val_lex (F.val = 3)
E | T | F | num | 3 + T | F | num | 7		T.val = 7 +.nada = -- E.val = 3
E E + T | T | F | num | 3 + 7		E.val = E_1.val + T.val (E.val = 3 + 7 = 10)

Como ya se ha adelantado, la pila además de tener estados y símbolos, también almacena intercaladamente los atributos.

4.2.3 Esquemas de traducción

Un esquema de traducción es una gramática de contexto libre en la que se encuentran intercalados, en el lado derecho de la regla de producción, fragmentos de programas a los que hemos llamado acciones semánticas. Un esquema de traducción es como una definición dirigida por sintaxis, con la excepción de que el orden de evaluación de las reglas semánticas se muestra explícitamente. La posición en la que se ejecuta alguna acción se da entre llaves y se escribe en el lado derecho de la regla de producción. Por ejemplo:

A → cd {printf(c + d)} B

Cuando se diseña un esquema de traducción, se deben respetar algunas limitaciones para asegurarse que el valor de un atributo esté disponible cuando una acción se refiera a él. Estas limitaciones, derivadas del tipo de análisis sintáctico escogido para construir el árbol, garantizan que las acciones no hagan referencia a un atributo que aún no haya sido calculado.

El ejemplo más sencillo ocurre cuando sólo se necesitan atributos sintetizados. En este caso, se puede construir el esquema de traducción creando una acción que conste de una asignación para cada regla semántica y colocando esta acción al final del lado derecho de la regla de producción asociada. Por ejemplo, la regla de producción $T \rightarrow T_1 * F$ y la acción semántica $T.val := T_1.val * F.val$ dan como resultado la siguiente producción y acción semántica:

$T \rightarrow T_1 * F \ \{T.val := T_1.val * F.val\}$

Si se tienen atributos tanto heredados como sintetizados, se debe ser más prudente, de manera que deben cumplirse algunas reglas a la hora de utilizar los atributos de los símbolos gramaticales:

1.- Un atributo heredado para un símbolo en el lado derecho de una regla de producción se debe calcular en una acción antes que dicho símbolo.

2.- Una acción no debe referirse a un atributo sintetizado de un símbolo que esté a la derecha de la acción.

3.- Un atributo sintetizado para el no terminal de la izquierda sólo se puede calcular después de que se hayan calculado todos los atributos a los que hace referencia. La acción que calcula dichos atributos se debe colocar, generalmente, al final del lado derecho de la producción.

Los esquemas de traducción bien definidos que cumplen estas restricciones se dice que están **bien definidos**. Ejemplo:

D → T {L.her = T.tipo} L ;
T → int {T.tipo = “integer”}
T → real { T.tipo = “real”}
$L \rightarrow \{L_1.her = L.her\}\ L_1$, id {añadetipo (id.ptr_tbs, L.her)}
L → id {añadetipo(id.ptr_tbs, L.her)}

4.2.4 Análisis LALR con atributos

En este epígrafe se estudiará cómo funciona el análisis LALR(1) con atributos. Recordemos que el LALR(1) es capaz de saber en todo momento qué regla aplicar sin margen de error (ante una gramática que cumpla las restricciones LALR(1)). Este hecho es el que se aprovecha para pasar a ejecutar las acciones de forma segura. En el caso de retroceso, sería posible que tras ejecutar una acción sea necesario retroceder porque nos hayamos dado cuenta de que esa regla no era la correcta. En tal caso, ¿cómo deshacer una acción?. Como ello es imposible, en un análisis con retroceso no es

posible ejecutar acciones a la vez que se hace el análisis, sino que habría que reconocer primero la sentencia, y en una fase posterior ejecutar las acciones con seguridad.

4.2.4.1 Conflictos reducir/reducir

Hay esquemas de traducción que no pueden ser reconocidos por analizadores LALR(1) directamente. En concreto, se suelen ejecutar acciones semánticas únicamente cuando se hace una reducción, y nunca cuando se desplaza. Por ello, un esquema de traducción con acciones intercaladas se convierte en otro equivalente que sólo contiene acciones al final de la regla, tal y como se ilustró en el epígrafe 4.1.2. A pesar de ello, esta transformación puede conducir a situaciones que dejan de ser LALR(1), como se ilustra en la figura 4.10.

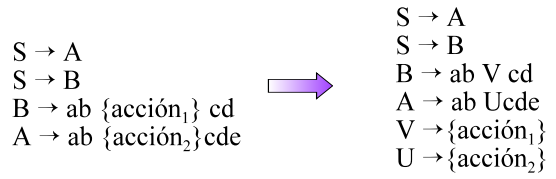


Figura 4.10. Ejemplo de gramática que no es LALR(1) después de haber sido transformada con reglas épsilon.

En este caso se produce un conflicto reducir/reducir, porque la gramática no es LALR.Como ya se ha indicado un conflicto reducir/reducir se produce cuando llega un momento del análisis en el que no se sabe que regla aplicar. En el caso anterior, si se introduce la cadena "abcd", cuando se hayan consumido los dos primeros *tokens* ("a" y "b") no podemos saber que regla ε aplicar, si U o V, ya que el *token* de prebúsqueda es "c" y éste sucede tanto a U como a V. Para solucionar este problema necesitaríamos un análisis LALR(3), ya que no es suficiente ver la "c"; tampoco es suficiente ver la "d"; es necesario llegar un carácter más allá, a la "e" o al EOF para saber por qué regla reducir y, por tanto, qué acción aplicar.

4.2.4.2 Conflictos desplazar/reducir

Hay casos más liosos, que se producen en gramáticas que son ambiguas, o sea, aquellas en que una sentencia puede tener más de un árbol sintáctico. Tal es el caso de una gramática tan sencilla como:

S → aBadd
S → aCd
B → a
C → aa

Al leer la cadena "aaad", cuando se consumen las dos primeras "a" y el *token* de prebúsqueda es la tercera "a", no se sabe si reducir la última "a" consumida a B o desplazar para reducir a continuación a C. Todo depende de con cuántas "d" finalice la cadena lo cual aún se desconoce.

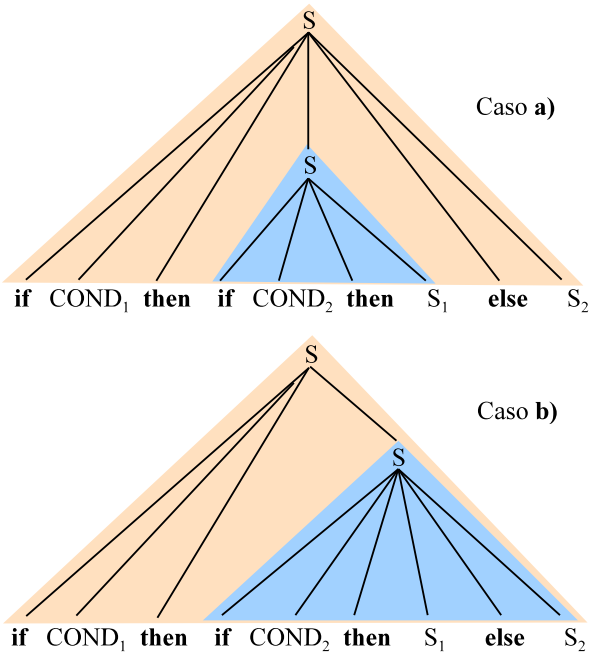


Figura 4.11. Ambigüedad en dos if anidados en C:
caso **a)** else asociado al if externo
caso **b)** else asociado al if interno

Un ejemplo de gramática a la que le sucede algo parecido es la correspondiente al reconocimiento de sentencias if anidadas en el lenguaje C. La gramática es:

S → if COND then S
S → if COND then S else S
S → id = EXPR

en la que se producen conflictos desplazar/reducir ante una entrada como:

if $COND_1$ **then if** $COND_2$ **then** S_1 **else** S_2

tal y como ilustra la figura 4.11.

Cuando un compilador de C se encuentra una sentencia como ésta opta por construir el segundo árbol sintáctico siguiendo la regla general "emparejar cada else con el then sin emparejar anterior más cercano". Esta regla para eliminar ambigüedades se puede incorporar directamente a la gramática añadiendo algunas reglas de producción adicionales:

S → SCompleta
| SIncompleta

SCompleta → if COND then SCompleta else SCompleta

	|	id = EXPR
SIncompleta	→	if COND then S
	|	if COND then SCompleta else SIncompleta

que asocian siempre el else al if más inmediato. De forma que una SCompleta es o una proposición if-then-else que no contenga proposiciones incompletas o cualquier otra clase de proposición no condicional (como por ejemplo una asignación). Es decir, dentro de un if con else solo se permiten if completos, lo que hace que todos los else se agrupen antes de empezar a reconocer los if sin else.

4.3 El generador de analizadores sintácticos PCYacc

PCYacc es un metacompilador que acepta como entrada una gramática de contexto libre, y produce como salida el autómata finito que reconoce el lenguaje generado por dicha gramática. Aunque PCYacc no acepta todas las gramáticas sino sólo las LALR (*LookAhead LR*), la mayoría de los lenguajes de programación se pueden expresar mediante una gramática de este tipo. Además permite asociar acciones a cada regla de producción y asignar un atributo a cada símbolo de la gramática (terminal o no terminal). Esto facilita al desarrollador la creación de esquemas de traducción en un reconocimiento dirigido por sintaxis. Dado que el autómata que genera PCYacc está escrito en lenguaje C, todas las acciones semánticas y declaraciones necesarias deben ser escritas por el desarrollador en este mismo lenguaje.

PCYacc parte de unos símbolos terminales (*tokens*), que deben ser generados por un analizador lexicográfico que puede implementarse a través de PCLex, o bien *ad hoc* siguiendo algunas convenciones impuestas por PCYacc.

4.3.1 Formato de un programa Yacc

Al lenguaje en el que deben escribirse los programas de entrada a PCYacc lo denominaremos Yacc. Así, un programa fuente en Yacc tiene la siguiente sintaxis:

```
Area de definiciones
%%
Area de reglas
%%
Area de funciones
```

Para ilustrar un programa fuente en YACC, partiremos de la siguiente gramática que permite reconocer expresiones aritméticas:

E → E + T | T
T → T * F | F
F → M (E) | M id | M num
M → - | ε

A continuación se describirán progresivamente cada una de estas áreas.

4.3.1.1 Área de definiciones

Este área está compuesta por dos secciones opcionales. La primera de ellas está delimitada por los símbolos **%{** y **%}** y permite codificar declaraciones ordinarias en C. Todo lo que aquí se declare será visible en las áreas de reglas y de funciones, por lo que suele usarse para crear la tabla de símbolos, o para incluirla (**#include**). Los símbolos **%{** y **%}** deben comenzar obligatoriamente en la columna 0 (al igual que todas las cláusulas que se verán a continuación)

La segunda sección permite declarar los componentes léxicos (*tokens*) usados en la gramática. Para ello se utiliza la cláusula **%token** seguida de una lista de terminales (sin separarlos por comas); siguiendo el ejemplo del punto 4.3.1 habría que declarar:

```
%token ID NUM
```

Además, a efectos de aumentar la legibilidad, pueden utilizarse tantas cláusulas **%token** como sean necesarias; por convención, los *tokens* se escriben en mayúsculas.

Internamente, PCYacc codifica cada *token* como un número entero, empezando desde el 257 (ya que del 0 al 255 se utilizan para representar los caracteres ASCII, y el 256 representa al *token* de error). Por lo tanto la cláusula

```
%token ID NUM
```

es equivalente a

```
# define ID nº1
# define NUM nº2
```

Evidentemente, los números o códigos asignados por PCYacc a cada *token* son siempre distintos.

No obstante lo anterior, no todos los terminales han de enumerarse en cláusulas **%token**. En concreto, los terminales se pueden expresar de dos formas:

- Si su lexema asociado está formado por un solo carácter, y el analizador léxico lo devuelve tal cual (su código ASCII), entonces se pueden indicar directamente en una regla de producción sin declararlo en una cláusula **%token**. Ejemplo: '+'.
- Cualquier otro terminal debe declararse en la parte de declaraciones, de forma que las reglas harán referencia a ellos en base a su nombre (no al código, que tan sólo es una representación interna).

Finalmente, en este área también puede declararse el axioma inicial de la gramática de la forma:

```
%start noTerminal
```

Si esta cláusula se omite, PCYacc considera como axioma inicial el antecedente de la primera regla gramatical que aparece en el área de reglas.

4.3.1.2 Creación del analizador léxico

Los analizadores léxicos los crearemos a través de la herramienta PCLex, y no *ad hoc*, con objeto de facilitar el trabajo. En el caso del ejemplo propuesto en el punto 4.3.1, se deberán reconocer los identificadores y los números, mientras que el

resto de los terminales, al estar formados por un solo carácter, pueden ser manipulados a través de su código ASCII. De esta manera, asumiendo que los espacios en blanco[2], tabuladores y retornos de carro actúan simplemente como separadores, disponemos de dos opciones principales:

```
%%      /* Opción 1 */
[0 - 9]+            { return NUM; }
[A- Z][A - Z0 -9]*  { return ID; }
[␣\t\n]             { ; }
.                   { return yytext[0]; }      /* Error sintáctico. Le pasa el error al a.si. */
```

y

```
%%      /* Opción 2 */
[0 - 9]+            { return NUM; }
[A- Z][A - Z0 -9]*  { return ID; }
[␣\t\n]             { ; }
'('|')'|'*'|'+'     { return yytext[0]; }      /* El a.si. sólo recibe tokens válidos */
.                   { printf ("Error léxico.\n"); }
```

Estas opciones se comportan de manera diferente ante los errores léxicos, como por ejemplo cuando el usuario teclea una '@' donde se esperaba un '+' o cualquier otro operador. En la Opción 1, el analizador sintáctico recibe el código ASCII de la '@' como *token*, debido al patrón **.**; en este caso el analizador se encuentra, por tanto, ante un *token* inesperado, lo que le lleva a abortar el análisis emitiendo un mensaje de error sintáctico. En la Opción 2 el analizador léxico protege al sintáctico encargándose él mismo de emitir los mensajes de error ante los lexemas inválidos; en este caso los mensajes de error se consideran lexicográficos; además, el desarrollador puede decidir si continuar o no el reconocimiento haciendo uso de la función **halt()** de C.

En los ejemplos que siguen nos decantamos por la Opción 1, ya que es el método del mínimo esfuerzo y concentra la gestión de errores en el analizador sintáctico.

Por último, nótese que el programa Lex no incluye la función **main()**, ya que PCYacc proporciona un modelo dirigido por sintaxis y es al analizador sintáctico al que se debe ceder el control principal, y no al léxico. Por tanto, será el área de funciones del programa Yacc quien incorpore en el **main()** una llamada al analizador sintáctico, a través de la función **yyparse()**:

```
%token ID NUM
%%
...
%%
void main ( ){
        yyparse( );
```

[2] El espacio en blanco se representará mediante el carácter "␣".

}

4.3.1.3 Área de reglas

Éste es el área más importante ya que en él se especifican las reglas de producción que forman la gramática cuyo lenguaje queremos reconocer. Cada regla gramatical tienen la forma:

noTerminal: consecuente ;

donde consecuente representa una secuencia de cero o más terminales y no terminales.

El conjunto de reglas de producción de nuestro ejemplo se escribiría en Yacc tal y como se ilustra en el cuadro 4.5 donde, por convención, los símbolos no terminales se escriben en minúscula y los terminales en mayúscula, las flechas se sustituyen por ':' y, además, cada regla debe acabar en ';'

Gramática	Notación Yacc
	%% /* área de reglas */
E → E + T	e : e '+' t
| T	| t
	;
T → T * F	t : t '*' f
| F	| f
	;
F → M (E)	f : m '(' e ')'
| M id	| m ID
| M num	| m NUM
	;
M → ε	m : /* Épsilon */
| -	| '-'
	;

Cuadro 4.5 Correspondencia entre una gramática formal y su equivalente en notación Yacc.

Junto con los terminales y no terminales del consecuente, se pueden incluir acciones semánticas en forma de código en C delimitado por los símbolos { y }. Ejemplo:

```
e    :    e '+' t    {printf("Esto es una suma.\n");};
```

Si hay varias reglas con el mismo antecedente, es conveniente agruparlas indicando éste una sola vez y separando cada consecuente por el símbolo '|', con tan sólo un punto y coma al final del último consecuente. Por lo tanto las reglas

gramaticales:

```
e   :   e '+' t   { printf("Esto es una suma.\n"); } ;
e   :   t         { printf("Esto es un término.\n"); } ;
```

se pueden expresar en Yacc como:

```
e   :   e '+' t   { printf("Esto es una suma.\n"); }
    |   t         { printf("Esto es un término.\n"); }
    ;
```

4.3.1.4 Área de funciones

La tercera sección de una especificación en Yacc consta de rutinas de apoyo escritas en C. Aquí se debe proporcionar un analizador léxico mediante una función llamada **yylex()**. En caso necesario se pueden agregar otros procedimientos, como rutinas de apoyo o resumen de la compilación.

El analizador léxico **yylex()** produce pares formados por un *token* y su valor de atributo asociado. Si se devuelve un *token* como por ejemplo NUM, o ID, dicho *token* debe declararse en la primera sección de la especificación en Yacc, El valor del atributo asociado a un *token* se comunica al analizador sintáctico mediante la variable **yylval**, como se verá en el punto 4.3.2. En caso de integrar Yacc con Lex, en esta sección habrá de hacer un **#include** del fichero generado por PCLex.

En concreto, en esta sección se deben especificar como mínimo las funciones:

- **main()**, que deberá arrancar el analizador sintáctico haciendo una llamada a **yyparse()**.
- **void yyerror(char * s)** que recibe una cadena que describe el error. En este momento la variable **yychar** contiene el terminal que se ha recibido y que no se esperaba.

4.3.2 Gestión de atributos

Como ya se ha indicado, en una regla se pueden incluir acciones semánticas, que no son más que una secuencia de declaraciones locales y sentencias en lenguaje C. Pero la verdadera potencia de las acciones en Yacc es que a cada terminal o no terminal se le puede asociar un atributo. Los nombres de atributos son siempre los mismos y dependen de la posición que el símbolo ocupa dentro de la regla y no del símbolo en sí; de esta manera el nombre **$$** se refiere al atributo del antecedente de la regla, mientras que **$i** se refiere al atributo asociado al i-ésimo símbolo gramatical del consecuente, es decir:

- El atributo del antecedente es **$$**.
- El atributo del 1er símbolo del consecuente es **$1**.
- El atributo del 2º símbolo del consecuente es **$2**.
- Etc.

La acción semántica se ejecutará siempre que se reduzca por la regla de producción asociada, por lo que normalmente ésta se encarga de calcular un valor para

$$ en función de los $i. Por ejemplo, si suponemos que el atributo de todos los símbolos de nuestra gramática (terminales y no terminales) es de tipo entero, se tendría:

```
%%
❶e  :   e '+' t     {$$ = $1 + $3;}
❷   |   t           {$$ = $1;}
    ;
❸t  :   t '*' f     { $$ = $1 * $3; }
❹   |   f           { $$ = $1; }
    ;
❺f  :   m '(' e ')'     { $$ = $1 * $3; }
❻   |   m ID            { $$ = $1 * $2; }
❼   |   m NUM           { $$ = $1 * $2; }
    ;
❽m  :   /* Épsilon */  { $$ = +1; }
❾   |   '-'             { $$ = -1; }
    ;
```

Por otro lado, en las reglas ❻ y ❼ del ejemplo anterior puede apreciarse que $2 se refiere a los atributos asociados a ID y NUM respectivamente; pero, al ser ID y NUM terminales, ¿qué valores tienen sus atributos?¿quién decide qué valores guardar en ellos?. La respuesta es que el analizador léxico es quien se tiene que encargar de asignar un valor a los atributos de los terminales. Recordemos que en el apartado 2.4.3.6 se estudió que PCLex proporcionaba la variable global **yylval** que permitía asociar información adicional a un *token*. Pues bien, mediante esta variable el analizador léxico puede comunicar atributos al sintáctico. De esta manera, cuando el analizador léxico recibe una cadena como "27+13+25", no sólo debe generar la secuencia de *tokens* NUM '+' NUM '+' NUM, sino que también debe cargar el valor del atributo de cada NUM, y debe de hacerlo antes de retornar el token (antes de hacer **return NUM;**). En nuestro caso asociaremos a cada NUM el valor entero que representa su lexema, con el objetivo de que las acciones semánticas del sintáctico puedan operar con él. El patrón Lex y la acción léxica asociada son:

```
[0-9]+   { yylval = atoi(yytext); return NUM; }
```

La función **atoi()** convierte de texto a entero. Nótese que siempre que se invoque a **atoi()**, la conversión será correcta, ya que el lexema almacenado en **yytext** no puede contener más que dígitos merced al patrón asociado. Además, se asume que la variable **yylval** también es entera.

PCYacc necesita alguna información adicional para poder realizar una correcta gestión de atributos, es decir, hemos de informarle del tipo de atributo que tiene asociado cada símbolo de la gramática, ya sea terminal o no terminal. Para ello debe definirse la macro **YYSTYPE** que indica de qué tipo son todos los atributos; en otras palabras, PCYacc asume que todos los atributos son de tipo **YYSTYPE**, por lo que hay que darle un valor a esta macro. De esta forma, si queremos que todos los atributos sean de tipo **int**, bastaría con escribir

```
%{
#define YYSTYPE int
```

```
%}
```

como sección de declaraciones globales en el área de definiciones. De hecho, PCYacc obliga a incluir en los programas Yacc una declaración como ésta incluso aunque no se haga uso de los atributos para nada: PCYacc exige asociar atributos siempre y conocer su tipo.

No obstante, resulta evidente que no todos los símbolos de una gramática necesitarán atributos del mismo tipo, siendo muy común que los números tengan asociado un atributo entero, y que los identificadores posean un puntero a una tabla de símbolos que almacene información importante sobre él. En este caso, puede declararse **YYSTYPE** como una **union** de C de manera que, aunque todos los símbolos de la gramática poseerán como atributo dicha **union**, cada uno utilizará sólo el campo que precise. Aunque este método puede llegar a derrochar un poco de memoria, permite una gestión uniforme de la pila de análisis sintáctico, lo que conlleva una mayor eficiencia. Es más, el mecanismo de usar una **union** como tipo de **YYSTYPE** es una práctica tan común que PCYacc posee una cláusula que permite definir **YYSTYPE** y declarar la **union** en un solo paso. La cláusula tiene la forma:

```
%union {
        /* Cuerpo de una union en C */
}
```

Recordemos que una **union** en C permite definir registros con parte variante. Es más, en una **union** todo es parte variante, o sea, todas los campos declarados en el cuerpo de una **union** comparten el mismo espacio de memoria, de manera que el tamaño de la **union** es el del campo más grande que contiene. Siguiendo con nuestro ejemplo, la unión tendría la forma:

```
%union {
        int numero;
        nodo * ptrNodo;
}
```

Con esto, **YYSTYPE** se ha convertido en una **union** (ya no es necesario incluir el #define YYSTYPE), de manera que cualquier símbolo puede hacer referencia al campo **numero** o al campo **ptrNodo**, pero no a ambos. Por regla general, cada símbolo gramatical, independientemente de la regla en que aparezca utiliza siempre el mismo campo del **%union**; por ello, PCYacc suministra un mecanismo para indicar cuál de los campos es el que se pretende utilizar para cada símbolo. Con esto además, nos ahorramos continuas referencias a los campos del **%union**. El mecanismo consiste en incluir entre paréntesis angulares en la cláusula %**token** del área de definiciones el campo que se quiere utilizar para cada terminal de la gramática; para indicar el tipo de los atributos de los no terminales, se utilizará la cláusula **%type** que, por lo demás, es igual a la **%token** excepto porque en ella se enumeran no terminales en lugar de terminales. Siguiendo esta notación, nuestro ejemplo quedaría:

```
%union {
        int numero;
```

```
            nodo * ptrNodo;
}
%token <numero> NUM
%token <ptrNodo> ID
%type <numero> e t f m
%%
❶e  :   e '+' t     {$$ = $1 + $3;}
❷   |   t           {$$ = $1;}
    ;
❸t  :   t '*' f     { $$ = $1 * $3; }
❹   |   f           { $$ = $1; }
    ;
❺f  :   m '(' e ')'     { $$ = $1 * $3; }
❻   |   m ID            { $$ = /*Extraer alguna información de (*$2) */; }
❼   |   m NUM           { $$ = $1 * $2; }
    ;
❽m  :   /* Épsilon */  { $$ = +1; }
❾   |   '-'             { $$ = -1; }
                ;
```

Nótese cómo, a pesar de que **YYSTYPE** es de tipo **union**, los $$ y $i no hacen referencia a sus campos, gracias a que las declaraciones **%token** y **%type** hacen estas referencias por nosotros. Es más, si en algún punto utilizamos el atributo de algún símbolo y no hemos especificado su tipo en una cláusula **%token** o **%type**, PCYacc emitirá un mensaje de error durante la metacompilación. Estas transformaciones son realizadas por PCYacc por lo que las acciones léxicas, gestionadas por PCLex, cuando hacen referencia a **yylval** deben indicar explícitamente el campo con que trabajar. El programa Lex que implementa el analizador léxico de nuestra gramática sería:

```
%%      /* Opción 1 */
[0 - 9]+            { yylval.numero = atoi(yytext); return NUM; }
[A- Z][A - Z0 -9]*  { yylval.ptrNodo = ...; return ID;}
[\b\t\n]            { ; }
.                   { return yytext[0]; }
```

La gestión del atributo asociado a los identificadores de usuario la dejaremos

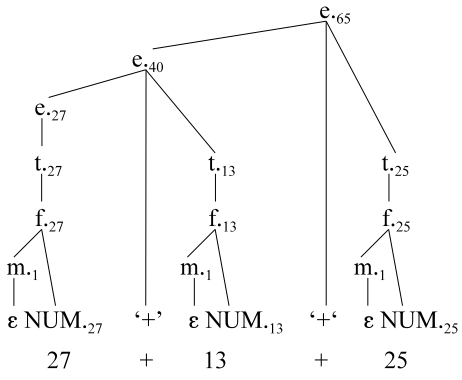


Figura 4.12. Árbol sintáctico para reconocer una cadena en la que no intervienen identificadores.

para el capítulo dedicado a la tabla de símbolos. Por ahora nos centraremos sólo en las constantes numéricas. La figura ? muestra el árbol generado para reconocer la cadena "27+13+25"; los valores de los atributos aparecen como subíndices.

4.3.2.1 Acciones intermedias

También es posible colocar acciones en medio de una regla, esto es, entre los símbolos de un consecuente. Sin embargo, antes se dijo que PCYacc construye analizadores que ejecutan una acción semántica cada vez que se reduce por su regla asociada en la construcción de un árbol sintáctico. Entonces, ¿cuándo se ejecuta una acción intermedia? En estos casos, PCYacc realiza una conversión de la gramática proporciona a otra equivalente en la que la acción intermedia aparece a la derecha de una nueva regla de la forma : $N_i \rightarrow \varepsilon$, donde **i** es un número arbitrario generado por PCYacc. Por ejemplo,

```
A : B { $$ = 1; } C { x = $2; y = $3; } ;
```

se convierte a

```
Ni : /* Épsilon */ { $$ = 1; } ;
A : B Ni C        {x = $2; y = $3;} ;
```

La ejecución de acciones semánticas está sujeta al mecanismo visto en el punto 4.1.2.

4.3.3 Ambigüedad

PCYacc no sólo permite expresar de forma fácil una gramática a reconocer y generar su analizador sintáctico, sino que también da herramientas que facilitan la eliminación de ambigüedades.

La gramática propuesta en el apartado 4.3.1 es un artificio para obligar a que el producto y la división tenga más prioridad que la suma y la resta, y que el menos unario sea el operador de mayor prioridad. Sin embargo, desde el punto de vista exclusivamente sintáctico, el lenguaje generado por dicha gramática resulta mucho más fácil de expresar mediante la gramática

```
e       :       e '+' e
        |       e '-' e
        |       e '*' e
        |       e '/' e
        |       '-' e
        |       '(' e ')'
        |       ID
        |       NUM
        ;
```

que, como resulta evidente, es ambigua. Por ejemplo, ante la entrada "27*13+25" se pueden obtener los árboles sintácticos de la figura 4.13; en el caso a) primero se multiplica y luego se suma, mientras que en el b) primero se suma y luego se multiplica. Dado que el producto tiene prioridad sobre la suma, sólo el caso a) es válido desde el punto de vista semántico, pues de otro modo se obtiene un resultado

incorrecto.

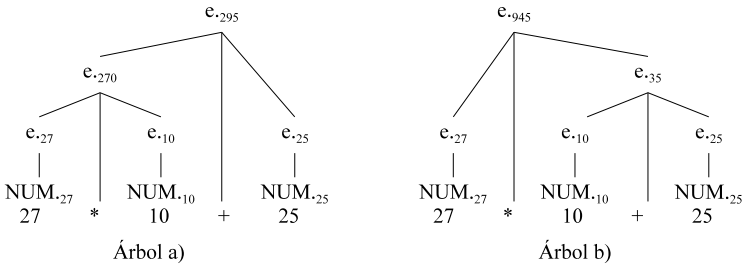


Figura 4.13. Árboles sintácticos que reconocen una sentencia válida según una gramática ambigua. Nótese cómo semánticamente se producen resultados diferentes según el orden de evaluación. La opción correcta es la a), ya que el producto tiene prioridad sobre la suma.

Esta situación se produce porque cuando el analizador tiene en la pila expr.$_{27}$ '*' expr.$_{10}$ y se encuentra con el '+', se produce un conflicto desplazar/reducir: si se desplaza se obtiene el árbol b), y si se reduce se obtiene el árbol a). Este mismo problema se habría producido en caso de tener dos sumas, en lugar de un producto y una suma; análogamente un árbol como el a) habría sido el correcto, ya que la suma es asociativa por la izquierda. En general, si una regla es recursiva por la izquierda implica que el operador que contiene es asociativo por la izquierda; e igual por la derecha. Por eso la gramática propuesta es ambigua, ya que cada regla es simultáneamente recursiva por la derecha y por la izquierda lo que implica que los operadores son asociativos por la izquierda y por la derecha a la vez, lo que no tiene sentido. Aunque la mayoría de operadores son asociativos a la izquierda, también los hay a la derecha, como es el caso del operador de exponenciación. Es más, algunos operadores ni siquiera son asociativos como puede ser el operador de módulo (resto) de una división.

Aunque ya hemos visto que este problema se puede solucionar cambiando la gramática, PCYacc suministra un mecanismo para resolver las ambigüedades producidas por conflictos del tipo desplazar/reducir sin necesidad de introducir cambios en la gramática propuesta. A menos que se le ordene lo contrario, PCYacc resolverá todos los conflictos en la tabla de acciones del analizador sintáctico utilizando las dos premisas siguientes:

1. Un conflicto reducir/reducir se resuelve eligiendo la regla en conflicto que se haya escrito primero en el programa Yacc.
2. Un conflicto desplazar/reducir se resuelve en favor del desplazamiento.

Además de estas premisas, PCYacc proporciona al desarrollador la posibilidad de solucionar por sí mismo los conflictos de la gramática. La solución viene de la mano de las cláusulas de precedencia y asociatividad: **%left**, **%right**, **%nonassoc** y **%prec**.

Las cláusulas **%left**, **%right** y **%nonassoc** permiten indicar la asociatividad de un operador (*left*-izquierda: reducir, *right*-derecha: desplazar y *nonassoc*-no asociativo: error); además, el orden en que se especifiquen las cláusulas permite a PCYacc saber qué operadores tienen mayor prioridad sobre otros. Estas cláusulas se colocan en el área de definiciones, normalmente después de la lista de terminales. Por ejemplo, la cláusula

```
%left '-'
```

hace que la regla e : e '-' e deje de ser ambigua. Nótese que para PCYacc no existe el concepto de operador (aritmético o no), sino que en estas cláusulas se indica cualquier símbolo en que se produzca un conflicto desplazar/reducir, ya sea terminal o no terminal.

Si se escriben varios símbolos en un mismo **%left**, **%right** o **%nonassoc** estaremos indicando que todos tiene la misma prioridad. Una especificación de la forma:

```
%left '-' '+'
%left '*' '/'
```

indica que '+' y '-' tienen la misma prioridad y que son asociativos por la izquierda y que '*' y '/' también. Como '*' y '/' van después de '-' y '+', esto informa a PCYacc de que '*' y '/' tienen mayor prioridad que '+' y '-'.

Hay situaciones en las que un mismo terminal puede representar varias operaciones, como en el caso del signo '-' que puede ser el símbolo de la resta o del menos unario. Dado que sólo tiene sentido que el '-' aparezca en una única cláusula resulta imposible indicar que, cuando se trata del menos binario la prioridad es inferior a la del producto, mientras que si es unario la prioridad es superior. Para solucionar esta situación, PCYacc proporciona el modificador **%prec** que, colocado al lado del consecuente de una regla, modifica la prioridad y asociatividad de la misma. **%prec** debe ir acompañado de un operador previamente listado en alguna de las cláusulas **%left**, **%right** o **%nonassoc**, del cual hereda la precedencia y la prioridad. En nuestro caso de ejemplo se podría escribir:

```
%left '+' '-'
%left '*' '/'
%%
e       :       e '+' e
        |       e '-' e
        |       e '*' e
        |       e '/' e
        |       '-' e      %prec '*'
        |       '(' e ')'
        |       ID
        |       NUM
        ;
```

indicándose de esta forma que la regla del menos unario tiene tanta prioridad como la del producto y que, en caso de conflicto, también es asociativa por la izquierda. No

obstante, esta solución presenta dos problemas: 1) la prioridad del menos unario no es igual a la del producto, sino superior; y 2) el menos unario no es asociativo por la izquierda, sino por la derecha (aunque esto no daría problemas, ya que no existe ambigüedad en la formación del árbol asociado a una sentencia como "---5", tal y como ilustra la figura 4.14).

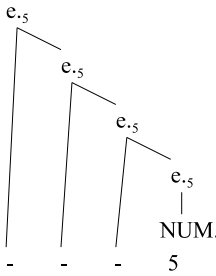


Figura 4.14. Aunque la gramática sea ambigua, si hay varios operadores "menos unario" seguidos, éstos sólo pueden asociarse a derecha.

Para resolver estos problemas, puede recurrirse a un artificio que consiste en declarar un terminal que representa a un operador ficticio de la prioridad y asociatividad requeridas con el único objetivo de usarlo posteriormente en un **%prec**. Este nuevo *token* no será retornado jamás por el analizador lexicográfico. Además, los terminales enumerados en una sentencia **%left**, **%right** o **%nonassoc** no hace falta declararlos también en el correspondiente **%token**, a no ser que posean un atributo del **%union**. Así, la solución final al problema de la prioridad y asociatividad viene dado por el bloque de programa Yacc:

```
%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO
%%
e       :       e '+' e
        |       e '-' e
        |       e '*' e
        |       e '/' e
        |       '-' e      %prec MENOS_UNARIO
        |       '(' e ')'
        |       ID
        |       NUM
        ;
```

4.3.4 Tratamiento de errores

Quizás una de las mayores deficiencias de PCYacc es la referente al tratamiento de errores ya que, por defecto, cuando el analizador sintáctico que genera se encuentra con un error sintáctico, sencillamente se para la ejecución con un parco mensaje "*syntax error*".

Esta ruptura de la ejecución puede evitarse mediante el correcto uso del *token* especial **error**. Básicamente, si se produce un error sintáctico el analizador generado por PCYacc desecha parte de la pila de análisis y parte de la entrada que aún le queda por leer, pero sólo hasta llegar a una situación donde el error se pueda recuperar, lo que se consigue a través del *token* reservado **error**. Un ejemplo típico del uso del *token* **error** es el siguiente:

```
prog    :       /* Épsilon */
        |       prog sent ';'
        |       prog error ';'
        ;
sent    :       ID '=' NUM
        ;
```

La última regla de este trozo de gramática sirve para indicarle a PCYacc que entre el no terminal **prog** y el terminal **';'** puede aparecer un error sintáctico. Supongamos que tenemos la sentencia a reconocer "a = 6; b = 7; c.= 8; d = 6;" en la que se ha "colado" un punto tras el identificador **c**; cuando el analizador se encuentre el *token* **'.'** se da cuenta de que hay un error sintáctico y busca en la gramática todas las reglas que contienen el símbolo **error**; a continuación va eliminando símbolos de la cima de la pila, hasta que los símbolos ubicados encima de la pila coincidan con los situados a la izquierda del símbolo **error** en alguna de las reglas en que éste aparece. A continuación va eliminando *tokens* de la entrada hasta encontrarse con uno que

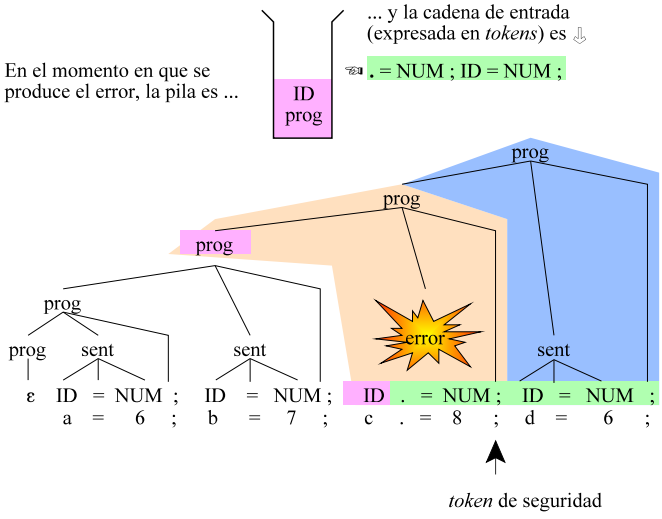


Figura 4.15. Recuperación de un error sintáctico. "Recuperar" no quiere decir "corregir", sino ser capaz de continuar el análisis sintáctico

coincida con el que hay justamente a la derecha en la regla de error seleccionada. Una vez hecho esto, inserta el *token* **error** en la pila, desplaza y continúa con el análisis intentando reducir por esa regla de error. Al terminal que hay justo a la derecha de **error** en una regla de error se le denomina *token* de seguridad, ya que permite al analizador llegar a una condición segura en el análisis. Como *token* de seguridad suele escogerse el de terminación de sentencia o de declaración, como en nuestro caso, en el que hemos escogido el punto y coma. La figura 4.15 ilustra gráficamente el proceso seguido para la recuperación del error.

Por último, para que todo funcione de forma adecuada es conveniente asociarle a la regla de error la invocación de una macro especial de PCYacc denominada **yyerrok**:

```
prog    :       /* Épsilon */
        |       prog sent ';'
        |       prog error ';'      { yyerrok; }
        ;
sent    :       ID '=' NUM
        ;
```

La ejecución de esta macro informa al analizador sintáctico de que la recuperación se ha realizado satisfactoriamente. Si no se invoca, el analizador generado por PCYacc entra en un estado llamado"condición de seguridad", y permanecerá en ese estado hasta haber desplazado tres *tokens*. La condición de seguridad previene al analizador de emitir más mensajes de error (ante nuevos errores sintácticos), y tiene por objetivo evitar errores en cascada debidos a una incorrecta recuperación del error. La invocación a **yyerrok** saca al analizador de la condición de seguridad, de manera que éste nos informará de todos, absolutamente todos, los errores sintácticos, incluso los debidos a una incorrecta recuperación de errores.

A pesar de poder recuperar los errores sintácticos, PCYacc no genera analizadores que informen correctamente del error, sino que solamente se limitan a informar de la existencia de éstos pero sin especificar su naturaleza. Esta falta se puede subsanar siguiendo los siguientes pasos:

1. En el área de funciones de Yacc, colocar la directiva:
   #include "errorlib.c"
   y no incluir la función **yyerror(char * s)**, que ya está codificada dentro del fichero **errorlib.c**.

2. Compilar el programa YACC como:
   pcyacc -d -pyaccpar.c fichero.yac
   -d crea el fichero **yytab.h** que tiene una descripción de la pila del analizador y el código asociado a cada *token*.
   -pyaccpar.c hace que se utilice un esqueleto para contener al autómata finito con pila que implementa PCYacc. En lugar de tomar el esqueleto por defecto, se toma a **yaccpar.c** que contiene las

nuevas funciones de tratamiento de errores. **yaccpar.c** hace uso, a su vez, del fichero **errorlib.h**.

3. Ejecutar el programa
   tokens.com
   Este programa toma el fichero **yytab.h**, y genera el fichero **yytok.h** que no es más que la secuencia de nombres de *tokens* entrecomillada y separada por comas. Este fichero es utilizado por **errorlib.c** para indicar qué *tokens* se esperaban.

4. Compilar **fichero.c** (por ejemplo, con Turbo C):
   tc fichero.c

De esta manera, cuando se ejecute **fichero.exe** y se produzca un error sintáctico, se obtendrá un clarificador mensaje de error:
"Error en la línea ...: encontrado el token ... y esperaba uno de los tokens ..."
donde los puntos suspensivos son rellenados automáticamente por el analizador sintáctico con la línea del error, el *token* incorrecto que lo ha provocado y la lista de *tokens* que habrían sido válidos, respectivamente.

4.3.5 Ejemplo final

Por último, uniendo todos los bloques de código Yacc que hemos ido viendo en los epígrafes anteriores, el ejemplo de la calculadora quedaría:

```
%{
#define YYSTYPE int
%}
%token NUMERO
%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO
%%
prog    :       /* Épsilon */
        |       prog expr ';'       { printf("= %d\n", $2); }
        |       prog error ';'      { yyerrok; }
        ;
expr    :       expr '+' expr       { $$ = $1 + $3; }
        |       expr '-' expr       { $$ = $1 - $3; }
        |       expr '*' expr       { $$ = $1 * $3; }
        |       expr '/' expr       { $$ = $1 / $3; }
        |       '-' expr   %prec MENOS_UNARIO   { $$ = - $2; }
        |       '(' expr ')'        { $$ = $2; }
        |       NUMERO              { $$ = $1; }
        ;
%%
#include "lexico.c"
void main() { yyparse(); }
void yyerror(char * s) { printf("%s\n", s); }
```

4.4 El generador de analizadores sintácticos Cup

Cup es una analizador sintáctico LALR desarrollado en el Instituto de Tecnología de Georgia (EE.UU.) que genera código Java y que permite introducir acciones semánticas escritas en dicho lenguaje. Utiliza una notación bastante parecida a la de PCYacc e igualmente basada en reglas de producción.

4.4.1 Diferencias principales

A modo de introducción, las principales diferencias con PCYacc, son:

- El antecedente de una regla se separa del consecuente mediante ::=, en lugar de mediante :.
- No es posible utilizar caracteres simples (´+´, ´-´, ´,´etc.) como terminales en una regla de producción, de manera que todos los terminales deben declararse explícitamente: MAS, MENOS, COMA, etc.
- En lugar de **%token** utiliza terminal.
- En lugar de **%type** utiliza non terminal.
- La lista de terminales se debe separar con comas y finalizar en punto y coma, al igual que la de no terminales. En ambas listas deben aparecer todos los terminales y no terminales, tanto si tienen atributo como si no.
- Si un terminal o no terminal tiene asociado un atributo, éste no puede ser de tipo primitivo sino que, obligatoriamente, debe ser un objeto (**Object**). Por tanto, en lugar de usar **int** se deberá usar **Integer**, en lugar de **char** se deberá usar **Character**, etc.
- No existe el **%union**, sino que se indica directamente el tipo de cada terminal y no terminal mediante el nombre de la clase correspondiente: **Integer**, **Character**, **MiRegistro**, etc. Tampoco es necesario el uso de corchetes angulares para indicar este tipo.
- En las acciones semánticas intermedias, los atributos del consecuente que preceden a la acción pueden usarse en modo sólo lectura (como si fueran del tipo **final** de Java).
- En lugar de **%left**, **%right** y **%nonassoc** se emplean, respectivamente precedence left, precedence right y precedence nonassoc. No obstante, la cláusula **%prec** sí se utiliza igual que en PCYacc.
- En lugar de **%start** se utiliza **start with**.
- No existen secciones que se deban separar por los símbolos %%.
- Pueden utilizarse, como si de un programa Java se tratase, las cláusulas **import** y **package**.

- Antes de empezar la ejecución del analizador generado puede ejecutarse un bloque de código Java mediante la cláusula:
  init with {: bloque_de_código_Java :};
- Para conectar el analizar generado con el analizador léxicográfico se utiliza la cláusula:
  scan with {: bloque_Java_que_retorna_un_objeto_de_la_clase_sym :};

Con estas pequeñas diferencias podemos ver a continuación cómo quedaría la especificación de nuestra calculadora:

```
package primero;
import java_cup.runtime.*;
/* Inicialización del analizador léxico (si es que hiciera falta) */
init with {: scanner.init(); :};
/* Solicitud de un terminal al analizador léxico. */
scan with {: return scanner.next_token(); :};
/* Terminales  */
terminal PUNTOYCOMA, MAS, MENOS, POR, ENTRE, MODULO;
terminal UMENOS, LPAREN, RPAREN;
terminal Integer NUMERO;
/* No terminales */
non terminal expr_list, expr_part;
non terminal Integer expr, term, factor;
/* Precedencias */
precedence left MAS, MENOS;
precedence left POR, ENTRE, MODULO;
precedence left UMENOS;
/* Gramática*/
lista_expr          ::=     lista_expr expr PUNTOYCOMA
                    |       /* Epsilon */
                    ;
expr        ::=     expr MAS expr
            |       expr MENOS expr
            |       expr POR expr
            |       expr ENTRE expr
            |       expr MODULO expr
            |       MENOS expr %prec UMENOS
            |       LPAREN expr RPAREN
            |       NUMERO
            ;
```

El propio Cup está escrito en Java, por lo que, para su ejecución, es necesario tener instalado algún JRE (*Java Runtime Environment*). La aplicación principal viene representada por la clase **java_cup.Main**, de tal manera que si el ejemplo de la calculado está almacenado en un fichero llamado **calculadora.cup**, se compilaría de la forma:

```
java java_cup.Main calculadora.cup
```

lo que produce los ficheros **sym.java** y **parser.java**. Como puede observarse, Cup no

respeta la convención (ampliamente aceptada) de nombrar con mayúsculas a las clases. La clase **sym.java** contiene las constantes necesarias para poder referirse a los terminales: sym.MAS, sym.POR, etc., lo que facilita el trabajo al analizador lexicográfico, que deberá retornar dichos terminales mediante sentencias de la forma: return new Symbol(sym.MAS); por ejemplo.

Para continuar nuestro ejemplo de la calculadora, es necesario incluir acciones semánticas y poder realizar referencias a los atributos tanto de los símbolos del consecuente, como del no terminal antecedente. De manera parecida a como sucede con Yacc, las acciones semánticas vienen encapsuladas entre los caracteres **{:** y **:}**. Sin embargo, la forma de referir los atributos difiere notablemente al método usado por Yacc.

Para empezar, el atributo del antecedente no se llama **$$**, sino **RESULT**; de manera similar, los atributos del consecuente no van numerados sino nombrados, lo que quiere decir que es el propio desarrollador quien debe decidir qué nombre se le va a dar al atributo de cada símbolo del consecuente. Este nombre se le indica a Cup colocándolo a continuación del terminal/no terminal en la regla de producción y separado de éste mediante el carácter dos puntos. De esta manera, a una regla como:

```
expr    ::=     expr MAS expr ;
```

se le daría significado semántico de la forma:

```
expr    ::=     expr:eIzq MAS expr:eDch
                {:RESULT = new Integer(eIzq.intValue() +
                                        eDch.intValue()); :}
        ;
```

Siguiendo este mecanismo, añadiendo el resto de acciones semánticas a nuestro ejemplo de la calculadora se obtiene:

```
import java_cup.runtime.*;
/* Inicialización del analizador léxico (si es que hiciera falta) */
init with {: scanner.init(); :};
/* Solicitud de un terminal al analizador léxico */
scan with {: return scanner.next_token(); :};
/* Terminales sin atributo */
terminal PUNTOYCOMA, MAS, MENOS, POR, ENTRE, MODULO;
terminal UMENOS, LPAREN, RPAREN;
/* Terminales con atributo asociado */
terminal Integer NUMERO;
/* No terminales sin atributo */
non terminal lista_expr;
/* No terminales con atributo asociado */
non terminal Integer expr;
/* Precedencias */
precedence left MAS, MENOS;
precedence left POR, ENTRE, MODULO;
precedence left UMENOS;
```

```
/* Gramática */
lista_expr    ::=    lista_expr expr:e PUNTOYCOMA  {: System.out.println("= " + e); :}
                |    lista_expr error PUNTOYCOMA
                |    /* Epsilon */
                ;
expr ::= expr:e1 MAS expr:e2
                {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
        |    expr:e1 MENOS expr:e2
                {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
        |    expr:e1 POR expr:e2
                {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
        |    expr:e1 ENTRE expr:e2
                {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
        |    expr:e1 MODULO expr:e2
                {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
        |    NUMERO:n
                {: RESULT = n; :}
        |    MENOS expr:e %prec UMENOS
                {: RESULT = new Integer(0 - e.intValue()); :}
        |    LPAREN expr:e RPAREN
                {: RESULT = e; :}
        ;
```

Como se ha dejado entrever anteriormente, para que todo esto funcione es necesario interconectarlo con un analizador lexicográfico que, ante cada terminal devuelva un objeto de tipo **java_cup.runtime.Symbol**. Los objetos de esta clase poseen un campo **value** de tipo **Object** que representa a su atributo, y al que el analizador lexicográfico debe asignarle un objeto coherente con la clase especificada en la declaración de terminales hecha en Cup.

4.4.2 Sintaxis completa.

Un fichero Cup está formado por cinco bloques principales que veremos a continuación.

4.4.2.1 Especificación del paquete e importaciones.

Bajo este primer bloque se indican opcionalmente las clases que se necesitan importar. También puede indicarse el paquete al que se quieren hacer pertenecer las clases generadas.

4.4.2.2 Código de usuario.

Cup engloba en una clase no pública aparte (incluida dentro del fichero **parser.java**) a todas las acciones semánticas especificadas por el usuario. Si el desarrollador lo desea puede incluir un bloque java dentro de esta clase mediante la declaración:

action code {: bloque_java :}

En este bloque pueden declararse variables, funciones, etc. todas de tipo

estático ya que no existen objetos accesibles mediante los que referenciar componentes no estáticos. Todo lo que aquí se declare será accesible a las acciones semánticas.

También es posible realizar declaraciones Java dentro de la propia clase **parser**, mediante la declaración:

parser code {: bloque_java :}

lo cual es muy útil para incluir el método **main()** que arranque nuestra aplicación.

Las siguientes dos declaraciones sirven para realizar la comunicación con el analizador léxicográfico. La primera es:

init with {: bloque_java :}

y ejecuta el código indicado justo antes de realizar la solicitud del primer *token*; el objetivo puede ser abrir un fichero, inicializar estructuras de almacenamiento, etc.

Por último, Cup puede comunicarse con cualquier analizador léxico, ya que la declaración:

scan with {: bloque_java :}

permite especificar el bloque de código que devuelve el siguiente *token* a la entrada.

4.4.2.3 Listas de símbolos

En este apartado se enumeran todos los terminales y no terminales de la gramática. Pueden tenerse varias listas de terminales y no terminales, siempre y cuando éstos no se repitan. Cada lista de terminales se hace preceder de la palabra **terminal**, y cada lista de no terminales se precede de las palabras **non terminal**. Los elementos de ambas listas deben estar separados por comas, finalizando ésta en punto y coma.

Con estas listas también es posible especificar el atributo de cada símbolo, ya sea terminal o no, sabiendo que éstos deben heredar de **Object** forzosamente, es decir, no pueden ser valores primitivos. Para ello, los símbolos cuyo tipo de atributo sea coincidente se pueden agrupar en una misma lista a la que se asociará dicho tipo justo detrás de la palabra **terminal**, de la forma:

terminal NombreClase terminal1, terminal2, etc.;

o bien

non terminal NombreClase noTerminal1, noTerminal2, etc.;

4.4.2.4 Asociatividad y precedencia.

Este apartado tiene la misma semántica que su homólogo en PCYacc, esto es, permite resolver los conflictos desplazar/reducir ante determinados terminales. De esta manera:

precedence left terminal1, terminal2, etc.;

opta por reducir en vez de desplazar al encontrarse un conflicto en el que el siguiente *token* es terminal1 o terminal2, etc. Por otro lado:

precedence right terminal1, terminal2, etc.;

opta por desplazar en los mismos casos. Finalmente:

precedence nonassoc terminal1, terminal2, etc.;

produciría un error sintáctico en caso de encontrarse con un conflicto desplazar/reducir en tiempo de análisis.

Pueden indicarse tantas cláusulas de este tipo como se consideren necesarias, sabiendo que el orden en el que aparezcan hace que se reduzca primero al encontrar los terminales de la últimas listas, esto es, mientras más abajo se encuentre una cláusula de precedencia, más prioridad tendrá a la hora de reducir por ella. De esta manera, es normal encontrarse cosas como:

precedence left SUMAR, RESTAR;
precedence left MULTIPLICAR, DIVIDIR;

lo que quiere decir que, en caso de ambigüedad, MULTIPLICAR y DIVIDIR tiene más prioridad que SUMAR y RESTAR, y que todas las operaciones son asociativas a la izquierda. Este tipo de construcciones sólo tiene sentido en gramáticas ambiguas, como pueda ser:

```
expr    ::=     expr MAS expr
        |       expr MENOS expr
        |       expr POR expr
        |       expr ENTRE expr
        |       LPAREN expr RPAREN
        |       NUMERO
        ;
```

Se asume que los símbolos que no aparecen en ninguna de estas listas poseen la prioridad más baja.

4.4.2.5 Gramática.

El último apartado de un programa Cup es la gramática a reconocer, expresada mediante reglas de producción que deben acabar en punto y coma, y donde el símbolo **::=** hace las veces de flecha.

La gramática puede comenzar con una declaración que diga cuál es el axioma inicial. Caso de omitirse esta cláusula se asume que el axioma inicial es el antecedente de la primera regla. Dicha declaración se hace de la forma:

start with noTerminal;

Al igual que en Yacc, es posible cambiar la prioridad y precedencia de una regla que produzca conflicto desplazar/reducir indicando la cláusula:

%prec terminal

junto a ella.

Las reglas de producción permiten albergar en su interior acciones semánticas, que son delimitadas por los símbolos **{:** y **:}**. En estas acciones puede colocarse cualquier bloque de código Java, siendo de especial importancia los accesos a los atributos de los símbolos de la regla. El atributo del antecedente viene dado por la variable Java **RESULT**, mientras que los atributos de los símbolos del consecuente son accesibles mediante los nombres que el usuario haya especificado para cada uno de

ellos. Este nombre se indica a la derecha del símbolo en cuestión y separado de éste por dos puntos. Su utilización dentro de la acción semántica debe ser coherente con el tipo asignado al símbolo cuando se lo indicó en la lista de símbolos. Debe prestarse especial atención cuando se utilicen acciones semánticas intermedias ya que, en tal caso, los atributos de los símbolos accesibles del consecuente sólo podrán usarse en modo lectura, lo que suele obligar a darles un valor inicial desde Jflex. Un ejemplo de esta situación podrá estudiarse en el apartado 8.4.6.

Por otro lado, y al igual que ocurría con PCYacc, el símbolo especial **error** permite la recuperación de errores mediante el método *panic mode*. En el apartado 7.4.4 se ilustra con más detalle la adecuada gestión de errores en Cup.

4.4.3 Ejecución de Cup.

El propio metacompilador Cup está escrito en Java, por lo que su ejecución obedece al modelo propio de cualquier programa Java. La clase principal es **java_cup.Main**, y toma su entrada de la entrada estándar:

```
java java_cup.Main < ficheroEntrada
```

Tambien es posible especificar algunas opciones al metacompilador, de la forma:

```
java java_cup.Main opciones < ficheroEntrada
```

donde las opciones más interesantes son:

-package nombrePaquete: las clases **parser** y **sym** se ubicarán en el paquete indicado.

- parser nombreParser: el analizador sintáctico se llamará **nombreParser** y no **parser**.

- symbols nombreSímbolos: la clase que agrupa a los símbolos de la gramática se llamará **nombreSímbolos** y no **sym**.

- expect numero: por defecto, Cup aborta la metacompilación cuando se encuentra con un conflicto reducir/reducir. Si se desea que el comportamiento se idéntico al de PCYacc, esto es, escoger la primera regla en orden de aparición en caso de dicho conflicto, es necesario indicarle a Cup exactamente el número de conflictos reducir/reducir que se esperan. Ello se indica mediante esta opción.

-progress: hace que Cup vaya informando a medida que va procesando la gramática de entrada.

-dump: produce un volcado legible de la gramática, los estados del autómata generado por Cup, y de las tablas de análisis. si se desea obtener sólo las tablas de análisis (utilizadas para resolver los conflictos), puede hacerse uso de la opción más concisa **-dump_states** que omite la gramática y las tablas de análisis.

Por otro lado, para que Cup se comunique convenientemente con el analizador léxico, éste debe implementar la interface **java_cup.runtime.Scanner**, definida como:

```
public interface Scanner {
        public Symbol next_token() throws java.lang.Exception;
}
```

Por último, para que nuestra aplicación funcione es necesario incluir una función **main()** de Java que construya un objeto de tipo **parser**, le asocie un objeto de tipo **Scanner** (que realizará el análisis lexicográfico), y arranque el proceso invocando a la función **parser()** dentro de una sentencia **try-catch**. Esto se puede hacer de la forma:

```
parser code {:
    public static void main(String[] arg){
        /* Crea un objeto parser */
        parser parserObj = new parser();
        /* Asigna el Scanner */
        Scanner miAnalizadorLexico =
                new Yylex(new InputStreamReader(System.in));
        parserObj.setScanner(miAnalizadorLexico);
      try{
            parserObj.parse();
        }catch(Exception x){
            System.out.println("Error fatal.");
        }
    }
:};
```

suponiendo que **Yylex** es la clase que implementa al analizador léxico.

4.4.4 Comunicación con JFlex

JFlex incorpora la opción **%cup** que equivale a las cláusulas:

```
%implements java_cup.runtime.scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
        return new java_cup.runtime.Symbol(sym.EOF);
%eofval}
%eofclose
```

Si el usuario ha cambiado el nombre de la clase que agrupa a los símbolos (por defecto **sym**), también deberá utilizar en JFlex la directiva:

```
%cupsym nombreSímbolos
```

para que **%cup** genere el código correctamente.

Como se desprende de las cláusulas anteriores, el analizador lexicográfico debe devolver los *tokens* al sintáctico en forma de objetos de la clase **Symbol**. Esta clase, que se encuentra en el paquete **java_cup.runtime** y que, por tanto, debe ser importada por el lexicográfico, posee dos constructores:

- Symbol(int n). Haciendo uso de las constantes enteras definidas en la clase

sym, este constructor permite crear objetos sin atributo asociado.

- Symbol(int n, Object o). Análogo al anterior, pero con un parámetro adicional que constituye el valor del atributo asociado al *token* a retornar.

De esta manera, el analizador léxico necesario para reconocer la gramática Cup que implementa nuestra calculadora, vendría dado por el siguiente programa JFlex:

```
import java_cup.runtime.*;
%%
%unicode
%cup
%line
%column
%%
"+"                 { return new Symbol(sym.MAS); }
"-"                 { return new Symbol(sym.MENOS); }
"*"                 { return new Symbol(sym.POR); }
"/"                 { return new Symbol(sym.ENTRE); }
"%"                 { return new Symbol(sym.MODULO); }
";"                 { return new Symbol(sym.PUNTOYCOMA); }
"("                 { return new Symbol(sym.LPAREN); }
")"                 { return new Symbol(sym.RPAREN); }
[:digit:]+          { return new Symbol(sym.NUMERO, new Integer(yytext())); }
[ \t\r\n]+  {;}
.                   { System.out.println("Error léxico."+yytext()+"-"); }
```

Capítulo 5

JavaCC

5.1 Introducción

JavaCC (*Java Compiler Compiler* - Metacompilador en Java) es el principal metacompilador en JavaCC, tanto por sus posibilidades, como por su ámbito de difusión. Se trata de una herramienta que facilita la construcción de analizadores léxicos y sintácticos por el método de las funciones recursivas, aunque permite una notación relajada muy parecida a la BNF. De esta manera, los analizadores generados utilizan la técnica descendente a la hora de obtener el árbol sintáctico.

5.1.1 Características generales

JavaCC integra en una misma herramienta al analizador lexicográfico y al sintáctico, y el código que genera es independiente de cualquier biblioteca externa, lo que le confiere una interesante propiedad de independencia respecto al entorno. A grandes rasgos, sus principales características son las siguientes:

- Genera analizadores descendentes, permitiendo el uso de gramáticas de propósito general y la la utilización de atributos tanto sintetizados como heredados durante la construcción del árbol sintáctico.
- Las especificaciones léxicas y sintácticas se ubican en un solo archivo. De esta manera la gramática puede ser leída y mantenida más fácilmente. No obstante, cuando se introducen acciones semánticas, recomendamos el uso de ciertos comentarios para mejorar la legibilidad.
- Admite el uso de estados léxicos y la capacidad de agregar acciones léxicas incluyendo un bloque de código Java tras el identificador de un token.
- Incorpora distintos tipos de tokens: normales (TOKEN), especiales (SPECIAL_TOKEN), espaciadores (SKIP) y de continuación (MORE). Ello permite trabajar con especificaciones más claras, a la vez que permite una mejor gestión de los mensajes de error y advertencia por parte de JavaCC en tiempo de metacompilación.
- Los tokens especiales son ignorados por el analizador generado, pero están disponibles para poder ser procesados por el desarrollador.
- La especificación léxica puede definir tokens de manera tal que no se diferencien las mayúsculas de las minúsculas bien a nivel global, bien en un patrón concreto.
- Adopta una notación BNF propia mediante la utilización de símbolos propios

de expresiones regulares, tales como (A)*, (A)+.

- Genera por defecto un analizador sintáctico LL(1). No obstante, puede haber porciones de la gramática que no sean LL(1), lo que es resuelto en JavaCC mediante la posibilidad de resolver las ambigüedades desplazar/desplazar localmente en el punto del conflicto. En otras palabras, permite que el a.si. se transforme en LL(k) sólo en tales puntos, pero se conserva LL(1) en el resto de las reglas mejorando la eficiencia.
- De entre los generadores de analizadores sintácticos descendentes, JavaCC es uno de los que poseen mejor gestión de errores. Los analizadores generados por JavaCC son capaces de localizar exactamente la ubicación de los errores, proporcionando información diagnóstica completa.
- Permite entradas codificadas en Unicode, de forma que las especificaciones léxicas también pueden incluir cualquier carácter Unicode. Esto facilita la descripción de los elementos del lenguaje, tales como los identificadores Java que permiten ciertos caracteres Unicode que no son ASCII.
- Permite depurar tanto el analizador sintáctico generado como el lexicográfico, mediante las opciones DEBUG_PARSER, DEBUG_LOOKAHEAD, y DEBUG_TOKEN_MANAGER.
- JavaCC ofrece muchas opciones diferentes para personalizar su comportamiento y el comportamiento de los analizadores generados.
- Incluye la herramienta JJTree, un preprocesador para el desarrollo de árboles con características muy potentes.
- Incluye la herramienta JJDoc que convierte los archivos de la gramática en archivos de documentación.
- Es altamente eficiente, lo que lo hace apto para entornos profesionales y lo ha convertido en uno de los metacompiladores más extendidos (quizás el que más, por encima de JFlex/Cup).

JavaCC está siendo mantenido actualmente por el grupo java.net de código abierto, y la documentación oficial puede encontrarse en el sitio https://javacc.dev.java.net. Por desgracia, dicha documentación es demasiado formal y no incorpora ningún tutorial de aprendizaje, aunque constituye la mejor fuente de información una vez familiarizado ligeramente con la herramienta, lo que es el objetivo del presente apartado.

5.1.2 Ejemplo preliminar.

Al igual que con Cup, comenzaremos nuestra andadura con un ejemplo preliminar que dé idea de la estructura de un programa de entrada a JavaCC y de la notación utilizada para ello. Se usará el nombre «JavaCC» para hacer referencia tanto a la herramienta JavaCC como al lenguaje que ésta admite para describir gramáticas.

Así, nuestro ejemplo escrito en JavaCC es:

```
options {
        LOOKAHEAD=1;
}
PARSER_BEGIN(Ejemplo)
        public class Ejemplo{
            public static void main(String args[]) throws ParseException {
                Ejemplo miParser = new Ejemplo(System.in);
                miParser .listaExpr();
            }
        }
PARSER_END(Ejemplo)

SKIP :{
            " "
        |   "\t"
        |   "\n"
        |   "\r"
}

TOKEN [IGNORE_CASE] : {
            <ID: (["a"-"z"])+>
        |   <NUM: (["0"-"9"])+>
}

void listaExpr() : {} {
            ( exprBasica() ";")+
}
void exprBasica() :{}{ LOOKAHEAD(2)
            <ID> "(" expr() ")"
        |   "(" expr() ")"
        |   "@NEW" <ID>
        |   <ID> "." <ID>
}
void expr() :{}{
            "@ALGO"
        |   <NUM>
}
```

Los programas JavaCC se suelen almacenar en ficheros con extensión **.jj**. Así, el ejemplo anterior se almacenaría en el fichero **Ejemplo.jj**, de manera que la metacompilación se hace invocando al fichero por lotes **javacc.bat** (quien a su vez invoca a la clase **javacc.class** del fichero **javacc.jar**) de la forma:

```
javacc Ejemplo.jj
```

lo que producirá los ficheros de salida:

- Ejemplo.java: es el analizador sintáctico.
- EjemploTokenManager.java: es el analizador lexicográfico.
- EjemploConstants.java: interface que asocia un código a cada token.

Asimismo, almacena en forma de String el nombre dado por el desarrollador a cada token, con objeto de que el analizador sintáctico pueda emitir mensajes de error claros y explicativos.

además de las clases necesarias:

- ParseException.java: clase que implementa los errores sintácticos. Cuando el analizador sintáctico encuentra uno de estos errores genera una excepción que puede ser capturada en cualquier nodo ancestro del árbol sintáctico que se estaba construyendo.
- SimpleCharStream.java: clase que implementa los canales de entrada de los cuales puede leer el analizador léxico. Incorpora constructores para convertir los canales de tipo tradicional: **InputStream** y **Reader**.
- Token.java: clase que implementa el objeto a través del cual se comunican el analizador léxico y el sintáctico.
- TokenMgrError.java: clase que implementa los errores lexicográficos. Este tipo de errores no se puede capturar de forma fácil, por lo que el desarrollador debe recoger todos los lexemas posibles y generar sus propios errores ante los incorrectos.

si no existían ya en el directorio desde el que se metacompila. El proceso puede apreciarse en la figura 5.1. Para compilar todos los ficheros producidos por JavaCC basta compilar el fichero principal de salida (**Ejemplo.java** según la figura) quien produce la compilación en cadena de todos los demás.

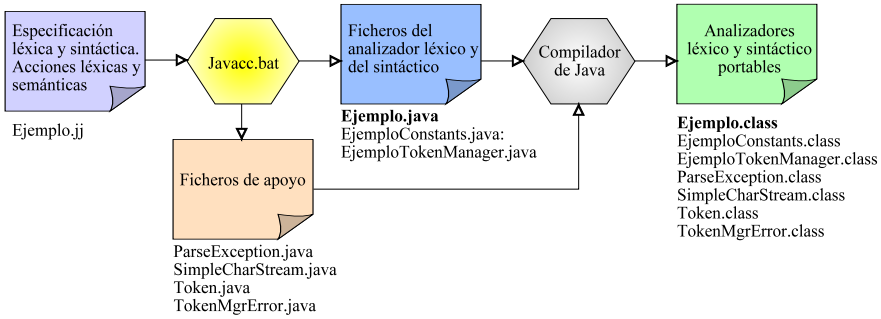


Figura 5.1. Proceso de metacompilación con JavaCC. Partiendo de un solo fichero (**Ejemplo.jj**), JavaCC produce 3 ficheros dependientes y otros 4 que son siempre idénticos.

5.2 Estructura de un programa en JavaCC

Como puede verse en el ejemplo propuesto, la estructura básica de un programa JavaCC es:

```
options {
    Área de opciones
}
PARSER_BEGIN(NombreClase)
    Unidad de compilación Java con la clase de nombre Nombreclase
```

PARSER_END(NombreClase)
Área de *tokens*
Área de funciones BNF

El área de opciones permite especificar algunas directrices que ayuden a JavaCC a generar analizadores léxico-sintácticos bien más eficientes, bien más adaptados a las necesidades concretas del desarrollador. En el ejemplo se ha indicado que, por defecto, la gramática indicada es de tipo LL(1), excepto si, en algún punto, se indica otra cosa.

Las cláusulas **PARSER_BEGIN** y **PARSER_END** sirven para indicarle a JavaCC el nombre de nuestra clase principal, así como para englobar tanto a ésta como a cualesquiera otras que se quieran incluir de apoyo, como pueda ser p.ej. un gestor de tablas de símbolos. En el ejemplo puede observarse que la clase principal constituye el analizador sintáctico en sí ya que la función **main()** crea un objeto de tipo **Ejemplo** a la vez que le pasa como parámetro en el constructor la fuente de la que se desea consumir la entrada: el teclado (**System.in**).

La clase creada por JavaCC incorporará una función por cada no terminal del área de reglas. Cada función se encargará de consumir la parte de la entrada que subyace debajo de su no terminal asociado en el árbol sintáctico de reconocimiento. Por tanto, asumiendo que el axioma inicial es **listaExpr**, una llamada de la forma **miParser.listaExpr()** consumirá toda la entrada, si ésta es aceptable.

Las siguientes dos áreas pueden mezclarse, aunque lo más usual suele ser indicar primero los *tokens* y finalmente las reglas en BNF, especialmente por motivos de claridad en el código.

En el ejemplo se han indicado *tokens* de dos tipos. Los *tokens* agrupados bajo la cláusula **SKIP** son aquellos que serán consumidos sin ser pasados al analizador sintáctico; en nuestro caso son: el espacio, el tabulador, el retorno de carro (CR-*Carry Return*) y la alimentación de línea (LF-*Line Feed*).

Los *tokens* bajo la cláusula **TOKEN** constituyen los *tokens* normales, aunque el desarrollador también puede indicar este tipo de *tokens* en las propias reglas BNF, como ocurre con los patrones "(", ")", ";", etc. La declaración de cada *token* se agrupa entre paréntesis angulares y está formada por el nombre del *token* seguido por el patrón asociado y separado de éste por dos puntos. Los patrones lexicográficos se describen de forma parecida a PCLex. El ejemplo ilustra el reconocimiento de un identificador formado sólo por letras (ya sean mayúsculas o minúsculas merced al modificador **[IGNORE_CASE]** de la cláusula **TOKEN**) y de un número entero.

La última área del ejemplo ilustra la creación de tres no terminales y sus reglas BNF asociadas. Dado que cada no terminal va a ser convertido por JavaCC en una función Java, su declaración es idéntica a la de dicha función y está sujeta a las mismas restricciones que cualquier otra función en Java. El cuerpo de cada una de estas funciones será construido por JavaCC y tendrá como propósito el consumo adecuado

de *tokens* en base a la expresión BNF que se indique en la especificación.

La notación BNF empleada hace uso del símbolo * que representa repetición 0 ó más veces, + para la repetición 1 ó más veces, | para la opcionalidad, paréntesis para agrupar, etc. Los terminales pueden indicarse de dos formas, bien colocando entre paréntesis angulares alguno de los declarados en la fase anterior, o bien indicando su patrón directamente si éste está formado por una cadena constante de caracteres (realmente, JavaCC permite cualquier expresión regular, pero ello no resulta útil para nuestros propósitos). En el ejemplo es de notar la inclusión del carácter "@" en los terminales *ad hoc* **NEW** y **ALGO**, ya que, en caso contrario serían considerados identificadores al encajar por el patrón del *token* **ID**. Los patrones "NEW" y "ALGO" habrían sido correctamente reconocidos si el *token* **ID** se hubiera declarado después de la aparición de éstos en las reglas BNF.

A continuación se estudiará con mayor detenimiento cada uno de estos apartados.

5.2.1 Opciones.

Esta sección comienza con la palabra reservada **options** seguida de una lista de una o más opciones entre llaves. La sección al completo puede omitirse, ya que no es obligatoria

Las opciones pueden especificarse tanto en el archivo de la gramática como en la línea de comandos. La misma opción no debe establecerse más de una vez, aunque en el caso de que se especifiquen en la línea de comandos, éstas tienen mayor prioridad que las especificadas en el archivo.

Los nombres de las opciones se escriben en mayúsculas, finalizan en ";" y se separan del valor que se les asocia mediante el signo "=". A continuación se detallan brevemente las opciones más importantes y las funciones que realizan:

LOOKAHEAD: indica la cantidad de *tokens* que son tenidos en cuenta por el analizador sintáctico antes de tomar una decisión. El valor por defecto es 1 lo que realizará análisis LL(1). Cuanto más pequeño sea el número de *tokens* de prebúsqueda, más veloz se realizarán los análisis. Este valor se puede modificar en determinadas reglas concretas.

CHOICE_AMBIGUITY_CHECK: sirve para que JavaCC proporcione mensajes de error más precisos cuando se encuentra con varias opciones con un prefijo de longitud igual o superior a la de **LOOKAHEAD**. En estos casos, JavaCC suele informar de que se ha encontrado un error y aconseja que se pruebe con un *lookahead* igual o superior a un cierto número **n**. Si **CHOICE_AMBIGUITY_CHECK** vale **k**, entonces JavaCC intentará averiguar el *lookahead* exacto que necesitamos (tan sólo a efectos de informarnos de ello) siempre y cuando éste no supere el valor **k**. La utilización de esta opción proporciona información más precisa, pero a costa

de un mayor tiempo de metacompilación. Su valor por defecto es 2.

FORCE_LA_CHECK: uno de los principales problemas que puede encontrarse JavaCC en la construcción del analizador sintáctico radica en la existencia de prefijos comunes en distintas opciones de una misma regla, lo que hace variar el número de *tokens* de prebúsqueda necesarios para poder escoger una de estas opciones. Cuando la opción **LOOKAHEAD** toma el valor 1, JavaCC realiza un control de errores mediante chequeos de prefijo en todas las opciones en las que el desarrollador no haya indicado un **LOOKEAHEAD** local, informando por pantalla de tales situaciones. Si **LOOKAHEAD** es superior a 1, entonces JavaCC no realiza dicho control en ninguna opción. Si se da a **FORCE_LA_CHECK** el valor *true* (por defecto vale *false*), el chequeo de prefijo se realizará en todas las opciones incluyendo las que posean **LOOKEAHEAD** local, a pesar de que éste sea correcto.

STATIC: si el valor es *true* (valor por defecto), todos los métodos y variables se crean como estáticas tanto en el analizador léxico como en el sintáctico lo que mejora notablemente su ejecución. Sin embargo, esto hace que sólo pueda realizarse un análisis cada vez e impide que, simultáneamente, distintos objetos de la misma clase puedan procesar diferentes fuentes de entrada. Si se define como estático, un mismo objeto puede procesar diferentes fuentes de manera secuencial, invocando **ReInit()** cada vez que quiera comenzar con una nueva. Si no es estático, pueden construirse (con **new**) diferentes objetos para procesar cada fuente.

DEBUG_PARSER: esta opción se utiliza para obtener información de depuración en tiempo de ejecución del analizador sintáctico generado. Su valor por defecto es *false*, aunque la traza de las acciones y decisiones tomadas puede activarse o desactivarse por programa invocando a **enable_tracing()** y **disable_tracing()** respectivamente. De manera parecida pueden utilizarse las opciones **DEBUG_LOOKAHEAD**, y **DEBUG_TOKEN_MANAGER**.

BUILD_PARSER: indica a JavaCC si debe generar o no la clase del analizador sintáctico. Su valor por defecto es *true*, al igual que el de la opción **BUILD_TOKEN_MANAGER**, lo que hace que se generen ambos analizadores.

IGNORE_CASE: si su valor es *true*, el analizador lexicográfico (*token manager*) generado no efectúa diferencia entre letras mayúsculas y minúsculas a la hora de reconocer los lexemas. Es especialmente útil en el reconocimiento de lenguajes como HTML. Su valor por defecto es *false*.

COMMON_TOKEN_ACTION: si el valor de esta opción es *true*, todas las invocaciones al método **getNextToken()** de la clase del analizador lexicográfico, causan una llamada al método **CommonTokenAction()** después del reconocimiento de cada lexema y después de ejecutar la acción

léxica asociada, si la hay. El usuario debe definir este último método en la sección **TOKEN_MGR_DECLS** del área de *tokens*. El valor por defecto es *false*.

UNICODE_INPUT: si se le da el valor *true* se genera un analizador léxico capaz de gestionar una entrada codificada en Unicode. Por defecto vale *false* lo que asume que la entrada estará en ASCII.

OUTPUT_DIRECTORY: es una opción cuyo valor por defecto es el directorio actual. Controla dónde son generados los archivos de salida.

5.2.2 Área de *tokens*

Este área permite la construcción de un analizador lexicográfico acorde a las necesidades particulares de cada proyecto. En la jerga propia de JavaCC los analizadores léxicos reciben el nombre de *token managers*.

JavaCC diferencia cuatro tipos de *tokens* o terminales, en función de lo que debe hacer con cada lexema asociado a ellos:

- SKIP: ignora el lexema.
- MORE: busca el siguiente el siguiente lexema pero concatenándolo al ya recuperado.
- TOKEN: obtiene un lexema y crea un objeto de tipo **Token** que devuelve al analizador sintáctico (o a quien lo haya invocado). Esta devolución se produce automáticamente sin que el desarrollador deba especificar ninguna sentencia **return** en ninguna acción léxica.
- SPECIAL_TOKEN: igual que SKIP pero almacena los lexemas de tal manera que puedan ser recuperados en caso necesario. Puede ser útil cuando se construyen traductores fuente-fuente de manera que se quieren conservar los comentarios a pesar de que no influyen en el proceso de traducción.

Una vez encontrado un lexema, sea cual sea su tipo, es posible ejecutar una acción léxica.

El formato de esta fase es el siguiente:

```
TOKEN_MGR_DECLS : {
    bloqueJava
}
<estadoLéxico> TOKEN [IGNORE_CASE] : {
        <token1 : patrón1> { acciónLéxica1 } : nuevoEstadoLéxico1
    |   <token2 : patrón2> { acciónLéxica2 } : nuevoEstadoLéxico2
    |       ...
}
<estadoLéxico> TOKEN [IGNORE_CASE] : {
        <token1 : patrón1> { acciónLéxica1 } : nuevoEstadoLéxico1
```

```
    |   <token2 : patrón2> { acciónLéxica2 } : nuevoEstadoLéxico2
    |       ...
}
...
```

donde:

- La palabra TOKEN puede sustituirse por SKIP, MORE o SPECIAL_TOKEN.
- La cláusula [IGNORE_CASE] es opcional pero, si se pone, no deben olvidarse los corchetes.
- El patrón puede ir precedido de un nombre de *token* y separado de éste por dos puntos, pero no es obligatorio (por ejemplo, en los separadores no tendría sentido).
- Si el patrón es una constante entrecomillada sin *token* asociado pueden omitirse los paréntesis angulares que lo engloban y, por tanto, el nombre.
- Los estados léxicos son identificadores cualesquiera, preferentemente en mayúsculas, o bien la palabra DEFAULT que representa el estado léxico inicial. Pueden omitirse. También puede indicarse una lista de estados léxicos separados por comas.
- Si se omite el estado léxico antes de la palabra TOKEN quiere decir que los patrones que agrupa pueden aplicarse en cualquier estado léxico.
- Si se omite el estado léxico tras un patrón quiere decir que, tras encontrar un lexema que encaje con él, se permanecerá en el mismo estado léxico.
- Las acciones léxicas están formadas por código Java y son opcionales.
- Las acciones léxicas tienen acceso a todo lo que se haya declarado dentro de la cláusula opcional TOKEN_MGR_DECLS.
- Las acciones léxicas se ejecutan en el seno de un *token manager* tras recuperar el lexema de que se trate.
- JavaCC sigue las mismas premisas que PCLex a la hora de buscar lexemas, esto es, busca siempre el lexema más largo posible y, en caso de que encaje por dos patrones, opta por el que aparezca en primer lugar.
- Si JavaCC encuentra un lexema que no encaja por ningún patrón produce un error léxico. Esta situación debe ser evitada por el desarrollador mediante el reconocimiento de los lexemas erróneos y la emisión del correspondiente mensaje de error.

5.2.2.1 Caracteres especiales para patrones JavaCC

Los caracteres que tienen un significado especial a la hora de formar expresiones regulares son:

"": sirve para delimitar cualquier cadena de caracteres.

<>: sirve para referenciar un *token* definido previamente. En algunas ocasiones puede resultar interesante crear *tokens* con el único objetivo de utilizarlos dentro de otros, como p.ej.:

```
TOKEN :
{
```

```
  < LITERAL_COMA_FLOTANTE:
       (["0"-"9"])+ "," (["0"-"9"])* (<EXPONENTE>)? (["f","F","d","D"])?
     | "." (["0"-"9"])+ (<EXPONENTE>)? (["f","F","d","D"])?
     | (["0"-"9"])+ <EXPONENTE> (["f","F","d","D"])?
     | (["0"-"9"])+ (<EXPONENTE>)? ["f","F","d","D"]
  >
|
  < EXPONENTE: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}
```

El problema de este ejemplo radica en que el analizador léxico puede reconocer la cadena "E123" como del tipo **EXPONENTE**, cuando realmente debiera ser un ID. En situaciones como ésta, el *token* **EXPONENTE** puede declararse como (nótese el uso del carácter "#"):

```
< #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
```

lo que le indica a JavaCC que no se trata de un *token* de pleno derecho, sino que debe utilizarlo como modelo de uso en aquellos otros patrones que lo referencien.

: indica repetición 0 ó mas veces de lo que le precede entre paréntesis. Ej.: ("ab") encaja con ε, "ab", "abab", etc. El patrón "ab"* es incorrecto puesto que el símbolo "*" debe estar precedido por un patrón entre paréntesis.

+: indica repetición 1 ó mas veces de lo que le precede entre paréntesis.

?: indica que lo que le precede entre paréntesis es opcional.

|: indica opcionalidad. Ej.: ("ab" | "cd") encaja con "ab" o con "cd".

[]: sirve para expresar listas de caracteres. En su interior pueden aparecer:

- Caracteres sueltos encerrados entre comillas y separados por comas. Ej.: ["a", "b", "c"] que encajará con los lexemas "a", "b" o "c".
- Rangos de caracteres formados por dos caracteres sueltos entre comillas y separados por un guión. Ej.: ["a"-"c"] que encajará con los lexemas "a", "b" o "c".
- Cualquier combinación de los anteriores. Ej.: ["a"-"z", "A"-"Z", "ñ", "Ñ"] que encajará con cualquier letra española sin signos diacríticos (acentos o diéresis).

~[]: sirve para expresar una lista de caracteres complementaria a la dada según lo visto anteriormente para []. P.ej., el patrón ~[] es una forma de emular al patrón (.|\n) de PCLex.

JavaCC siempre crea automáticamente el *token* **<EOF>** que encaja con el carácter fin de fichero.

Por otro lado, no existe una área de declaración de estados léxicos, como en PCLex o JFlex, sino que éstos se utilizan directamente. Un sencillo ejemplo para reconocer comentarios no anidados al estilo de Java o C sería:

```
SKIP : {
        "/*" : DentroComentario
}
<DentroComentario> SKIP : {
        "*/" : DEFAULT
}
<DentroComentario> MORE : {
        <~[]>
}
```

5.2.2.2 Elementos accesibles en una acción léxica

El desarrollador puede asociar acciones léxicas a un patrón que serán ejecutadas por el analizador léxico cada vez que encuentre un lexema acorde. En estas acciones se dispone de algunas variables útiles, como son:

image: variable de tipo **StringBuffer** que almacena el lexema actual. Una copia de dicho lexema se almacena también en el campo **token** de la variable **matchedToken**. Si el último *token* aceptado fue de tipo **MORE**, entonces **image** acumulará el lexema actual a lo que ya contuviese. P.ej. si tenemos:

```
<DEFAULT> MORE : {
        "a" {     ❶ ;} : S1
}
<S1> MORE : {
        "b" {     ❷
                  int long = image.length()-1;
                  image.setCharAt(long, image.charAt(long).toUpperCase());
                  ❸
        } : S2
}
<S2> TOKEN : {
        "cd" {    ❹ ; } : DEFAULT
}
```

ante la entrada “abcd”, la variable **image** tendrá los valores “a”, “ab”, “aB” y “aBcd” en los puntos ❶, ❷, ❸ y ❹ del código respectivamente.

lenghtOfMatch: variable de tipo entero que indica la longitud del último lexema recuperado. Siguiendo el ejemplo anterior, esta variable tomaría los valores 1, 1, 1 y 2 en los puntos ❶, ❷, ❸ y ❹ del código respectivamente. La longitud completa de **image** puede calcularse a través del método **length()** de la clase **StringBuffer**.

curLexState: variable de tipo entero que indica el estado léxico actual. Los estados léxicos pueden ser manipulados directamente a través del nombre que el desarrollador les halla dado, ya que JavaCC inicializa las correspondientes constantes Java en uno de los ficheros que genera (**EjemploConstants.java** según el ejemplo del apartado 5.1.2).

matchedToken: variable de tipo **Token** que almacena el *token* actual. Siguiendo

el ejemplo anterior, **matchedToken.token** tomaría los valores "a", "ab", "ab" y "abcd" en los puntos ❶, ❷, ❸ y ❹ del código respectivamente.

switchTo(int estadoLéxico): función que permite cambiar por programa a un nuevo estado léxico. Si se invoca esta función desde una acción léxica debe ser lo último que ésta ejecute. Evidentemente, para que tenga efecto no debe usarse la cláusula **: nuevoEstadoLéxico1** tras la acción léxica. Esta función puede resultar muy útil en el reconocimiento, p. ej., de comentarios anidados:

```
TOKEN_MGR_DECLS : {
        int numComentarios = 0;
}
SKIP: {
        "/*" { numComentarios= 1;} : DentroComentario
}
<DentroComentario> SKIP: {
                "/*" { numComentarios++;}
        |       "*/" { numComentarios--; if (numComentarios == 0)
                switchTo(DEFAULT); }
        |       <(~[\*\\\f\t\n\r])+>
        |       <[\*\\\f\t\n\t]>
}
```

Las acciones léxicas también pueden utilizar todo lo que se haya declarado en la cláusula opcional **TOKEN_MGR_DECLS**. el siguiente ejemplo visualiza la longitud de una cadena entrecomillada haciendo uso de varios patrones y una variable común **longCadena** (nótese la importancia del orden de los dos últimos patrones ya que, de otra forma nunca se reconocerían las comillas de cierre):

```
TOKEN_MGR_DECLS : {
        int longCadena;
}
MORE : {
        "\"" { longCadena= 0;} : DentroCadena
}
<DentroCadena> TOKEN : {
        <STRLIT: "\""> {System.out.println("Size = " + longCadena);} :
DEFAULT
}
<DentroCadena> MORE : {
        <~["\n","\r"]> {longCadena++;}
}
```

Si la opción **COMMON_TOKEN_ACTION** es *true* debe especificarse obligatoriamente la clásula **TOKEN_MGR_DECLS** que, como mínimo, deberá contener la función **CommonTokenAction()**, cuya cabecera es:

```
void CommonTokenAction(Token t)
```

y que será invocada cada vez que se recupere cualquier lexema sea cual sea su tipo, tal y como se vio en el apartado 5.2.1.

5.2.2.3 La clase Token y el *token manager*

Como ya se ha comentado, cada vez que el analizador léxico encuentra un lexema correspondiente a un *token* normal, retorna un objeto de la clase **Token**. Las variables y funciones principales de esta clase son:

- **image**: variable de tipo **String** que almacena el lexema reconocido.
- **kind**: para cada patrón al que el desarrollador le asocia un nombre, JavaCC incluye una constante de tipo entera en la clase de constantes que genera. La variable **kind** almacena uno de dichos enteros, concretamente el del nombre del patrón por el que encajó su lexema.
- **beginLine**, **beginColumn**, **endLine** y **endColumn**: variables de tipo entero que indican, respectivamente la línea y columna de comienzo y de final del lexema.
- **specialToken**: variable de tipo **Token** que referencia al token anterior si y sólo si éste era de tipo especial; en caso contrario almacenará el valor **null**.

Para finalizar, indicar que JavaCC proporciona en la clase del analizador léxico (**EjemploTokenManager** según el ejemplo del punto 5.1.2) dos métodos que pueden ser interesantes, caso de querer construir tan sólo dicho analizador, a saber:

- Un constructor que toma como parámetro un objeto de tipo **CharStream** del que tomará la entrada a analizar.
- La función **getNextToken()** que devuelve el siguiente objeto de tipo **Token** que encuentra en la entrada. El desarrollador debe controlar el tipo de *token* **<EOF>** para no consumir más allá del fin de la entrada.

El siguiente ejemplo ilustra un programa JavaCC del que sólo se pretende emplear la parte de analizador léxico. A pesar de que la opción **BUILD_PARSER** está a *false*, JavaCC obliga a indicar las cláusulas **PARSER_BEGIN** y **PARSER_END**, y una clase del mismo nombre en su interior, aunque esté vacía.

```
options{
    /* No generar la clase Ejemplo */
    BUILD_PARSER=false;
}
PARSER_BEGIN(Ejemplo)
    public class Ejemplo{}
PARSER_END(Ejemplo)
/* Declaración de la función main() en el token manager */
TOKEN_MGR_DECLS : {
    public static void main(String args[]) throws ParseException {
        EjemploTokenManager miLexer =
            new EjemploTokenManager(new SimpleCharStream(System.in));
        Token t;
        while((t=miLexer.getNextToken()).kind != EjemploTokenManager.EOF)
```

```
                System.out.println("Vd. ha escrito"+t.image);
    }
}
SKIP : {
        " "
    |   "\t"
    |   "\n"
    |   "\r"
}
TOKEN [IGNORE_CASE] : {
    <ID: (["a"-"z"])+>
}
```

De esta manera, la función **main()** se incluye directamente en el *token manager* mediante la cláusula **TOKEN_MGR_DECLS**, y en ella construimos un canal de entrada del tipo esperado por el analizador léxico, a quien se lo pasamos en su constructor. Los canales de entrada de los que puede leer el *token manager* son del tipo **CharStream**; JavaCC proporciona la clase **SimpleCharStream** que hereda de **CharStream** para facilitar la creación de canales aceptables para ser analizados lexicográficamente. **SimpleCharStream** es un envoltorio o estrato para las clases **InputStream** y **Reader** poseyendo constructores que toman como parámetros objetos de cualesquiera de estos dos tipos.

Una vez hecho esto, los *tokens* se van recuperando de uno en uno visualizando por pantalla el correspondiente mensaje. El mismo efecto se podría haber obtenido escribiendo:

```
options{ BUILD_PARSER=false; }
PARSER_BEGIN(Ejemplo) public class Ejemplo{} PARSER_END(Ejemplo)
TOKEN_MGR_DECLS : {
  public static void main(String args[]) throws ParseException {
    new EjemploTokenManager(new SimpleCharStream(System.in)).getNextToken();
  }
}
SKIP : {
    " "
  | "\t"
  | "\n"
  | "\r"
}
SKIP [IGNORE_CASE] : {
  <ID: (["a"-"z"])+> { System.out.println("Vd. ha escrito"+image); }
}
```

mucho más compacto y donde se han resaltado las modificaciones con respecto al equivalente anterior.

5.2.3 Área de funciones BNF

Como ya se ha comentado, JavaCC genera un analizador sintáctico

descendente implementado a base de funciones recursivas, de manera que cada no terminal de nuestra gramática será convertido una función diferente, cuya implementación será generada por JavaCC.

Este área tiene la siguiente estructura:

$tipoRetorno_1$ $funcionJava_1(param_1)$:
    { $codigoJava_1$ }
    { $exprBNF_1$ }
$tipoRetorno_2$ $funcionJava_2(param_2)$:
    { $codigoJava_2$ }
    { $exprBNF_2$ }
    …

Dado que cada no terminal se convertirá en una función en Java, el desarrollador no sólo debe indicar su nombre, sino la cabecera completa de dicha función. Este hecho es de fundamental importancia puesto que:

> «JavaCC permite el intercambio de atributos entre reglas BNF mediante el paso de parámetros y la obtención de resultados en el momento de hacer uso de un no terminal (o lo que es lo mismo, invocar a la función que lo implementa) lo que equivale, respectivamente, a enviar atributos hacia abajo y hacia arriba en el árbol sintáctico».

Por otro lado, las expresiones BNF **$exprBNF_i$** pueden hacer uso de los siguientes componentes:

noTerminal(): representa la utilización del no terminal **noTerminal**. Nótese la utilización de los paréntesis de invocación de función.

<>: permiten hacer referencia a un terminal declarado en el área de *tokens*. Ej.: <NUMERO>

“”: permiten expresar *tokens* anónimos, entrecomillando un literal de tipo cadena de caracteres.

*: indica repetición 0 ó mas veces de lo que le precede entre paréntesis.

+: indica repetición 1 ó mas veces de lo que le precede entre paréntesis.

?: indica que lo que le precede entre paréntesis es opcional.

[]: tiene igual significado que el ?, de manera que (patron)? es equivalente a [patron].

|: indica opcionalidad. Ej.: (“texto1” | “texto2”) encaja con “texto1” y con “texto2”.

De esta manera, una forma para reconocer expresiones vendría dada por la gramática:

```
void expr():{}{
    term() ( (“+”|”-“) term() )*
```

```
}
void term():{}{
    fact() ( ("*"|"/") fact() )*
}
void fact():{}{
            <NUMERO>
    |       <ID>
    |       "(" expr() ")"
}
```

El apartado **codigoJava**$_i$ permite ubicar declaraciones y sentencias Java al comienzo de la implementación de una función no terminal. Entre otras cosas, esto nos puede permitir obtener una traza de las acciones que toma nuestro analizador sintáctico para reconocer una sentencia. Por ejemplo, el programa:

```
void expr():{ System.out.println("Reconociendo una expresión"); }{
        term() ( ("+"|"-") term() )*
}
void term():{ System.out.println("Reconociendo un término"); }{
        fact() ( ("*"|"/") fact() )*
}
void fact():{ System.out.println("Reconociendo un factor"); }{
                <NUMERO>
        |       <ID>
        |       "(" expr() ")"
}
```

ante la entrada "23+5*(8-5)" produciría la salida:

```
Reconociendo una expresión
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un factor
Reconociendo una expresión
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
```

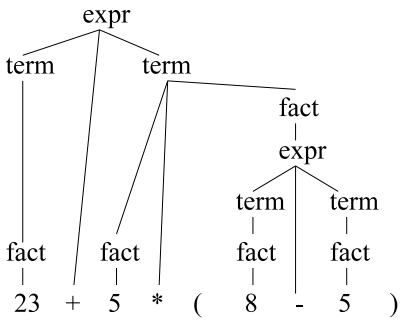


Figura 5.2. Árbol sintáctico que reconoce una expresión en base a reglas BNF. Para mayor claridad no se han dibujado los *tokens* NUMERO

lo que coincide con el recorrido en preorden del árbol sintáctico que reconoce dicha sentencia, y que se muestra en la figura 5.2.

Nótese que cuando se utiliza notación BNF, el número de símbolos bajo una misma regla puede variar si en ésta se emplea la repetición. P.ej., siguiendo con la gramática anterior, la regla de la **expr** puede dar lugar a tantos hijos como se quiera en el árbol sintáctico; sirva como demostración el árbol asociado al reconocimiento de "23+5+7+8+5", que se ilustra en la figura 5.3, y cuya corroboración viene dada por la salida del analizador sintáctico:

```
Reconociendo una expresión
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
```

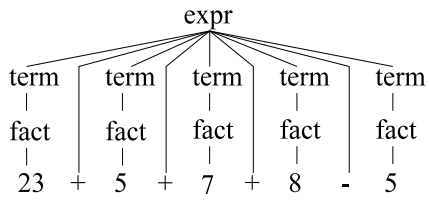


Figura 5.3. El árbol sintáctico asociado a una expresión BNF con expansiones del tipo * ó + puede tener un número indeterminado de hijos

Como ya se ha comentado, JavaCC genera por defecto analizadores de tipo LL(1), lo que quiere decir que son capaces de escoger entre diferentes opciones de flujo consultando tan sólo el siguiente *token* de la entrada. Sin embargo, ya sabemos que no todas las gramáticas son LL(1), por lo que puede haber situaciones en las que sea necesario consultar más de un *token* de la entrada para poder tomar una decisión. Por ejemplo, la siguiente gramática:

```
void expr():{}{
                <ID>
        |       <ID> "." expr()
        |       <ID> "()"
}
```

no es LL(1) porque, ante una entrada como "cont.aux()" el analizador sintáctico se encuentra como primer *token* con un **<ID>**, pero como las tres posibles opciones de **expr()** comienzan por **<ID>**, pues se carece de suficiente información para saber por cuál de ellas optar. En este caso, una prebúsqueda de dos *tokens* nos permitiría decidir correctamente.

Para ello, la opción **LOOKAHEAD** vista en el apartado 5.2.1 permite indicarle a JavaCC el tipo de gramática que se le está proporcionando, esto es, el número de *tokens* que debe prebuscar antes de tomar una decisión.

No obstante lo anterior, en la mayoría de los casos, las gramáticas de los lenguajes de

programación contienen una mayoría de expresiones LL(1), y tan sólo unas pocas reglas para las cuales puede ser necesario una prebúsqueda de 2, o 3 a lo sumo. En estos casos, se obtendrá un analizador sintáctico más eficiente si se deja la prebúsqueda por defecto (a 1), y se le indica a JavaCC en qué casos concretos debe emplear una prebúsqueda superior. Esto se consigue con la prebúsqueda o *lookahead* local, que tiene la forma:

LOOKAHEAD(**n**) $patron_1$

y que es considerado en sí mismo un patrón también. El significado de esta cláusula es muy sencillo: le indica al analizador sintáctico que, antes de decidirse a entrar por el **$patron_1$** se asegure de que éste es el correcto tomando **n** *tokens*. De esta manera, la gramática anterior se puede dejar como LL(1) e indicar la prebúsqueda local de la forma:

```
void expr():{}{
                    LOOKAHEAD(2) <ID>
          |         LOOKAHEAD(2) <ID> "." expr()
          |         <ID> "()"
}
```

Además, el propio JavaCC es capaz de decidir si la prebúsqueda especificada por el desarrollador es adecuada o no, informando de ello en caso negativo tanto por exceso como por defecto.

5.3 Gestión de atributos.

JavaCC carece de una gestión de atributos específica, pero permite el paso de parámetros y la devolución de valores cada vez que se invoca a un no terminal, tal y como se introdujo en el punto 5.2.3. De esta manera, cada vez que se invoca a un no terminal, podemos asignar el valor que devuelve a una variable global declarada en el bloque de código Java asociado a dicha regla; este valor hace las veces de atributo sintetizado. Además, cuando se declara un no terminal, puede poseer parámetros formales, de tal manera que deba ser invocada con parámetros reales que harán las veces de atributos heredados.

Desgraciadamente, JavaCC tampoco permite asociar atributos a los terminales, por lo que deberán ser las acciones semánticas quienes trabajen directamente con los lexemas. Aunque esto puede ser un engorro en la mayoría de los casos, tiene la ventaja de que cada acción semántica puede extraer del lexema la información que le convenga, en lugar de utilizar siempre el mismo atributo en todos los consecuentes en los que parezca un mismo terminal. El siguiente ejemplo ilustra las reglas BNF y las acciones semánticas asociadas para crear una calculadora:

```
int expr():{
      int  acum1=0,
           acum2=0;
}{
      acum1=term()     (    ❶("+" acum2=term() {acum1+=acum2;} )
                        |   ❷("-" acum2=term() {acum1-=acum2;} )
                        )*
      { return acum1; }
}
int term():{
      int  acum1=0,
           acum2=0;
}{
      acum1=fact()     (    ❸("*" acum2=fact() {acum1*=acum2;} )
```

```
                        |   ❹("/" acum2=fact() {acum1/=acum2;} )
                        )*
        { return acum1; }
}
int fact():{
        int acum=0;
}{
            <NUMERO>            { return Integer.parseInt(token.image); }
        |   "(" acum=expr() ")"  { return acum; }
}
```

Como puede observarse en este ejemplo, cuando un no terminal devuelve algún valor, debe incluirse una acción semántica al final de cada una de las opciones que lo definen, con una sentencia **return** coherente. Además, es posible colocar acciones semánticas en el interior de las subexpresiones BNF (como en el caso de **expr** y **term**, en los puntos ❶, ❷, ❸ y ❹). Una acción semántica se ejecutará cada vez que el flujo de reconocimiento pase por ellas. Por ejemplo, ante la entrada 23+5+11 se pasará 2 veces por el punto ❶, ya que es necesario pasar 2 veces por la subexpresión ("+" term())* para reconocer dicha entrada.

Además, nótese que una acción semántica colocada justo después de un terminal puede acceder al objeto *token* asociado al mismo, a través de la variable **token**. Esto es así a pesar de la prebúsqueda que se haya indicado, ya que es el analizador sintáctico el que carga dicha variable justo después de consumir cada terminal, e independientemente de los que haya tenido que prebuscar para tomar una decisión sobre qué opción tomar.

Cuando se incluyen acciones semánticas en el interior de una expresión BNF, ésta se enrarece sobremanera, siendo muy difícil reconocer la gramática en un mar de código. Para evitar este problema se recomienda colocar en un comentario delante de cada no terminal la expresión BNF pura que lo define.

Por otro lado, cuando conviene que un terminal **t** tenga siempre el mismo atributo en todos los consecuentes en los que aparece, suele resultar más legible crear una regla **T::=t**, asociarle a **T** el atributo calculado a partir de **t** y, en el resto de reglas, usar **T** en lugar de **t**.

Siguiendo los dos consejos anteriores, el ejemplo de la calculadora quedaría:

```
/*
expr ::= term ( ( "+" | "-" ) term )*
*/
int expr():{
        int   acum1=0,
              acum2=0;
}{
        acum1=term()    (   ❶("+" acum2=term() {acum1+=acum2;} )
                        |   ❷("-" acum2=term() {acum1-=acum2;} )
                        )*
        { return acum1; }
```

```
}
/*
term ::= fact ( ( "*" | "/" ) fact )*
*/
int term():{
      int   acum1=0,
            acum2=0;
}{
      acum1=fact()      (     ❸("*" acum2=fact() {acum1*=acum2;} )
                        |     ❹("/" acum2=fact() {acum1/=acum2;} )
                        )*
      { return acum1; }
}
/*
fact ::= NUMERO | "(" expr ")"
*/
int fact():{
      int acum=0;
}{
            acum=numero()         { return acum; }
      |     "(" acum=expr() ")"   { return acum; }
}
int numero(): {}{
      <NUMERO> { return Integer.parseInt(token.image); }
}
```

5.4 Gestión de errores.

La gestión de errores puede conseguirse en JavaCC en dos vertientes:

- Personalizando los mensajes de error.
- Recuperándose de los errores.

5.4.1 Versiones antiguas de JavaCC

Hasta la versión 0.6 de JavaCC para personalizar los mensajes de error era necesario establecer la opción **ERROR_REPORTING** a *true*. Con ello JavaCC proveerá a la clase del analizador sintáctico con las variables:

- **error_line**, **error_column**: variables de tipo entero que representan la línea y la columna en la que se ha detectado el error.
- **error_string**: variable de tipo **String** que almacena el lexema que ha producido el error. Si la prebúsqueda (*lookahead*) se ha establecido a un valor superior a 1, **error_string** puede contener una secuencia de lexemas.
- **expected_tokens**: variable de tipo **String[]** que contiene los *tokens* que habrían sido válidos. De nuevo, si la prebúsqueda se ha establecido a un valor superior a 1, cada ítem de **expected_tokens** puede ser una secuencia de *tokens*

en lugar de un *token* suelto.

Cada vez que el analizador sintáctico se encuentre un error sintáctico invocará a la función **token_error(),** que deberá ser re-escrita por el desarrollador por lo que éste deberá crearse una clase nueva que herede de la clase del analizador sintáctico.

De manera parecida, en el analizador léxico pueden hacerse uso de los elementos:

- **error_line**, **error_column**: línea y columna del error léxico.
- **error_after**: cadena de caracteres que contiene el trozo de lexema que se intentaba reconocer.
- **curChar**: carácter que ha producido el error léxico.
- **LexicalError()**: función invocada por el *token manager* cuando se encuentra un error léxico.

5.4.2 Versiones modernas de JavaCC

Actualmente, JavaCC gestiona los errores léxicos y sintácticos lanzando las excepciones **TokenMgrError** y **ParseException**, cuyas clases genera JavaCC automáticamente si no existen ya en el directorio de trabajo. La filosofía es que el desarrollador debe impedir que se produzcan errores léxicos, controlando adecuadamente cualquier lexema de entrada inclusive los no válidos, en cuyo caso deberá emitir el correspondiente mensaje de error.

De esta manera, el desarrollador puede modificar la clase **ParseException** para adaptarla a sus necesidades. Por defecto, esta clase posee las variables:

- **currentToken**: de tipo **Token**, almacena el objeto *token* que ha producido el error.
- **expectedTokenSequences**: de tipo **int[][]** tiene el mismo objetivo que la variable **expected_tokens** vista en el apartado anterior.
- **tokenImage**: de tipo **String[]** lmacena los últimos lexemas leídos. Si la prebúsqueda está a 1, entonces se cumple:

```
tokenImage[0].equals(currentToken.image);
```

Por otro lado, la recuperación de errores no requiere modificación alguna de la clase **ParseException**, sino que para ello se utiliza una cláusula similar a la **try-catch** de Java para capturar la excepción en el punto que más convenga. Dicha cláusula se considera asimismo un patrón, por lo que puede aparecer en cualquier lugar donde se espere uno. La cláusula es de la forma:

```
try {
        patron
        catch(CualquierExcepcion x){
                codigoDeRecuperacionJava
```

```
        }
    }
```

Nótese que la cláusula **try-catch** de JavaCC puede capturar cualquier excepción, y no sólo las de tipo **ParseException**, por lo que el desarrollador puede crear sus propias excepciones y elevarlas dónde y cuando se produzcan. Por ejemplo, puede resultar útil el crearse la excepción **TablaDeSimbolosException**.

El código de recuperación suele trabajar en *panic mode*, consumiendo toda la entrada hasta llegar a un *token* de seguridad. El siguiente ejemplo ilustra cómo recuperar errores en sentencias acabadas en punto y coma.

```
    void sentenciaFinalizada() : {} {
        try {    ( sentencia() <PUNTOYCOMA> )
        catch (ParseException e) {
            System.out.println(e.toString());
            Token t;
            do {
                t = getNextToken();
            } while (t.kind != PUNTOYCOMA);
        }
    }
```

5.5 Ejemplo final

Ensamblando las distintas partes que se han ido viendo a lo largo de este capítulo, y observando los consejos dados en cuanto a comentarios, el ejemplo completo de la calculadora con control de errores quedaría:

```
PARSER_BEGIN(Calculadora)
    public class Calculadora{
      public static void main(String args[]) throws ParseException {
    new Calculadora(System.in).gramatica();
    }
    }
PARSER_END(Calculadora)
SKIP : {
        " "
    |   "\t"
    |   "\n"
    |   "\r"
}
TOKEN [IGNORE_CASE] :
{
        <NUMERO: (["0"-"9"])+>
    |   <PUNTOYCOMA: ";">
}
/*
gramatica ::= ( exprFinalizada )+
*/
void gramatica():{}{
```

```
      ( exprFinalizada() )+
}
/*
exprFinalizada ::= expr ";"
*/
void exprFinalizada():{
      int resultado;
}{
      try{
            resultado=expr() <PUNTOYCOMA> {System.out.println("="+resultado); }
      }catch(ParseException x){
            System.out.println(x.toString());
            Token t;
            do {
                  t = getNextToken();
            } while (t.kind != PUNTOYCOMA);
      }
}
/*
expr ::= term ( ( "+" | "-" ) term )*
*/
int expr():{
      int   acum1=0,
            acum2=0;
}{
      acum1=term()      (     ("+" acum2=term() {acum1+=acum2;} )
                        |     ("-" acum2=term() {acum1-=acum2;} )
                        )*
      { return acum1; }
}
/*
term ::= fact ( ( "*" | "/" ) fact )*
*/
int term():{
      int   acum1=0,
            acum2=0;
}{
      acum1=fact()      (     ("*" acum2=fact() {acum1*=acum2;} )
                        |     ("/" acum2=fact() {acum1/=acum2;} )
                        )*
      { return acum1; }
}
/*
fact ::= NUMERO | "(" expr ")"
*/
int fact():{
      int   acum=0,
            uMenos=1;
}{
```

```
    ("-" { uMenos = -1; } )?
    (    <NUMERO>            { return Integer.parseInt(token.image)*uMenos; }
     |   "(" acum=expr() ")"  { return acum*uMenos; }
    )
}
SKIP : {
    <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
}
```

Nótese cómo se ha tenido que definir el terminal **PUNTOYCOMA** para poder hacer referencia a él en la gestión del error.

Por otro lado, el orden de las expresiones regulares es fundamental, de tal manera que la regla:

```
<ILEGAL: (~[])>
```

recoge todos los lexemas que no hayan entrado por ninguna de las reglas anteriores, incluidas las *ad hoc* especificadas en la propia gramática BNF.

5.6 La utilidad JJTree

JJTree es el preprocesador para JavaCC. Inserta acciones que construyen árboles parser en el fuente JavaCC. La salida generada por JJTree se ejecuta a través de JavaCC para crear el parser.

5.6.1 Características

Por defecto, JJTree genera código para construir el árbol de nodos del parse para cada no terminal de nuestro lenguaje. Este funcionamiento puede modificarse de manera que ciertos no terminales no generen nodos, o generar un nodo a partir de una expansión de una regla de producción.

JJTree define una interfaz de Java tipo **Node,** el cual todos los nodos del árbol parser deben implementar. La interfaz proporciona métodos para operaciones como inicialización del tipo del padre de un nodo generado, así como adición y recuperación de hijos.

JJTree puede operar de dos formas distintas: simple o multi (para buscar términos mejores). En modo simple, cada nodo del árbol parser es del tipo **SimpleNode**; en modo multi, el tipo de los nodos del árbol parse se deriva del nombre del nodo. Si no se proporciona implementaciones para la clase nodo, JJTree generará implementaciones basadas en la clase **SimpleNode**, pudiendo modificar las implementaciones de manera que puedan sernos de utilidad.

5.6.2 Ejemplo Práctico

Al final del presente documento hay una sección (Pasos de Ejecución) en la que se indica cuáles son los pasos que hay que dar para la compilación de ficheros **.jjt** y **.jj.** Un avance se muestra a continuación.

Para la ejecución de JJTree es necesario alimentar la entrada de la siguiente

manera:

```
jjtree ejemplo1.jjt
```

Todos los archivos que se generan de forma automática se pueden consultar en la sección Ficheros Fuente.

Supongamos que creamos un archivo llamado **ejemplo1.jjt** con las siguientes especificaciones:

```
PARSER_BEGIN(ejemplo1)
class ejemplo1 {
	public static void main(String args[]) {
		System.out.println("Entrada teclado");
		ejemplo1 ins = new ejemplo1(System.in);
		try {
			SimpleNode nodo = ins.Inicio();
			nodo.dump("");
		} catch (Exception e) {
			System.out.println(e.getMessage());
			e.printStackTrace();
		}
	}
}
PARSER_END(ejemplo1)
SKIP :{
		" "
	|	 "\t"
	|	 "\n"
	|	 "\r"
	|	 <"//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
	|	 <"/*" (~["*"])* "*" (~["/"] (~["*"])* "*")* "/">
}

TOKEN : /* LITERALES */
{
<ENTERO:
	<DECIMAL>
	|	<HEX>
	|	<OCTAL>
>
|
  < #DECIMAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL: "0" (["0"-"7"])* >
}

TOKEN : /* IDENTIFICADORES */
{
```

```
  < IDENTIFICADOR: <LETRA> (<LETRA>|<DIGITO>)* >
|
  < #LETRA: ["_","a"-"z","A"-"Z"] >
|
  < #DIGITO: ["0"-"9"] >
}

SimpleNode Inicio() : {}
{
        Expresion() ";"
        { return jjtThis; }
}
void Expresion() : {}
{
        ExpSuma()
}
void ExpSuma() : {}
{
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
void ExpMult() : {}
{
        ExpUnaria() ( ( "*" | "/" | "%" ) ExpUnaria() )*
}
void ExpUnaria() : {}
{
         "(" Expresion() ")" | Identificador() | <ENTERO>
}
void Identificador() : {}
{
        <IDENTIFICADOR>
}
```

La sentencia **e.printStackTrace()**; me indica que obtengo el contenido de la pila cuando ha ocurrido un error. Una traza de ejemplo se muestra a continuación; en este caso, el error ocurrido es el acceso a una zona de memoria que está a null:

```
...
java.lang.NullPointerException
        at ejemplo4.Identificador(ejemplo4.java:228)
        at ejemplo4.ExpUnaria(ejemplo4.java:200)
        at ejemplo4.ExpMult(ejemplo4.java:139)
        at ejemplo4.ExpSuma(ejemplo4.java:84)
        at ejemplo4.Expresion(ejemplo4.java:75)
        at ejemplo4.ExpUnaria(ejemplo4.java:196)
        at ejemplo4.ExpMult(ejemplo4.java:139)
        at ejemplo4.ExpSuma(ejemplo4.java:84)
        at ejemplo4.Expresion(ejemplo4.java:75)
        at ejemplo4.Inicio(ejemplo4.java:45)
        at ejemplo4.Main(ejemplo4.java:30)
```

Por otro lado, **dump()** es un método de la clase **SimpleNode** que se genera automáticamente, y que hace posible un recorrido automático por todos los nodos hijo del padre que se le pasa como parámetro. La cadena de caracteres pasada al método indica el prefijo que se le añade a los nodos cuando se muestran por pantalla. Todo esto se explica a lo largo del documento.

Si ejecutamos **jjtree ejemplo1.jjt** obtenemos cuatro ficheros:

- **Node.java**,
- **SimpleNode.java**,
- **ejemplo1TreeConstants.java** y
- **JJTejemplo1State.java**

El contenido de **ejemplo1TreeConstants.java** es el siguiente:

```
public interface ejemplo1TreeConstants {
        public int JJTINICIO = 0;
        public int JJTEXPRESION = 1;
        public int JJTEXPSUMA = 2;
        public int JJTEXPMULT = 3;
        public int JJTEXPUNARIA = 4;
        public int JJTIDENTIFICADOR = 5;
        public String[] jjtNodeName = {
                "Inicio",
                "Expresion",
                "ExpSuma",
                "ExpMult",
                "ExpUnaria",
                "Identificador",
        };
}
```

Una vez hecho esto, si queremos que nuestra gramática reconozca una sentencia introducida por la entrada asignada en la cabecera del fuente (en este caso el teclado), hemos de ejecutar el siguiente comando

javacc ejemplo1.jj

El archivo **ejemplo1.jj** ha sido generado de forma automática al reconocer JJTree nuestra gramática como correcta.

Los archivos generados automáticamente por **javacc** para cualquier fuente **.jj** están especificados en la sección **Ficheros Fuente**.

Compilamos con **javac** el fichero .java que hemos obtenido. Dicho ejecutable lo encontramos en la versión 1.5 de JDK de Java. Tras este paso, obtenemos el ejecutable que lanzará la lectura de la cadena a reconocer:

javac ejemplo1.java
java ejemplo1

Si le proporcionamos la entrada siguiente

(id + id) * (id + id)

el árbol generado es el siguiente:

```
Inicio
Expresion
  ExpSuma
    ExpMult
      ExpUnaria
        Expresion
          ExpSuma
            ExpMult
              ExpUnaria
                Identificador
            ExpMult
              ExpUnaria
            Identificador
              ExpUnaria
                Expresion
                  ExpSuma
                    ExpMult
                      ExpUnaria
                        Identificador
                    ExpMult
                      ExpUnaria
                        Identificador
```

Vemos que se le asigna un valor de tipo entero a cada no-terminal (función java en el archivo **.jjt**) de forma única, y que por defecto, se le asigna un nombre a cada nodo (no-terminal). Dicho nombre, si no es asignado por el usuario, es el mismo que el nombre del método que implementa al no-terminal. Para poder asignarle un nombre distinto al que se le pone por defecto, es necesario utilizar el carácter # seguido del nombre que le queremos asignar.

5.6.2.1 Renombrado de nodos mediante

es un constructor que se utiliza para que el usuario pueda llamar a cada nodo de una forma distinta a como JavaCC le llama por defecto. Ha de operar en modo multi para que cada nodo pueda derivarse del nombre que pongamos inmediatamente después del constructor #.Sirve también para simplificar la sintaxis y eliminar de la pila los nodos que no aportan información al árbol generado por nuestra gramática.

En la zona de opciones, el **modo multi** debe estar **activo** de la siguiente manera:

```
options {
        MULTI=true;
}
```

Si utilizamos esto último para el mismo ejemplo (**ejemplo1bis.jjt**), la sintaxis sería:

```
void Expresion() #void : {}
```

```
{
        ExpSuma()
}
void ExpSuma() #void : {}
{
        (ExpMult() ( ( "+" | "-" ) ExpMult() )*)#Suma(>1)
}
void ExpMult() #void: {}
{
        (ExpUnaria()(("*"|"/"|"%")ExpUnaria())*)#Mult(>1)
}
void ExpUnaria() #void : {}
{
        "(" Expresion() ")" | Identificador() | <LITERAL_ENTERO>
}
```

Y el archivo **ejemplo1bisTreeConstans** quedaría:

```
public interface ejemplo1bisTreeConstants {
        public int JJTINICIO = 0;
        public int JJTVOID = 1;
        public int JJTSUMA = 2;
        public int JJTMULT = 3;
        public int JJTIDENTIFICADOR = 4;
        public String[] jjtNodeName = {
                "Inicio",
                "void",
                "Suma",
                "Mult",
                "Identificador",
        };
}
```

Alimentando de nuevo la entrada con la misma cadena, obtenemos la siguiente salida: (traza de la pila)

```
Inicio
Mult
        Suma
                Identificador
                Identificador
        Suma
                Identificador
                Identificador
```

Observamos que aquellos no-terminales a los que hemos renombrado como **#void,** no aparecen en la salida correspondiente a la pila. Esto es muy útil cuando queremos utilizar sólo los nodos que realmente aportan información al árbol, como ya se ha comentado anteriormente.

5.6.2.2 Construcción del Árbol

Aunque JavaCC es un generador de gramáticas descendente, JJTree construye el árbol de la secuencia de reglas gramaticales desde abajo, es decir, de forma ascendente. Para ello, hace uso de una pila donde se almacenan los nodos una vez generados. Cuando se intenta reducir un conjunto de nodos mediante la asignación de un nodo padre para todos ellos, dichos nodos hijos se sacan de la pila y se le asocia al padre, almacenando posteriormente el conjunto padre-hijos en la pila. Siempre que la pila esté accesible (pila abierta) se podrá efectuar operaciones en el ámbito de las acciones gramaticales tales como 'push','pop' y cualquier otra que se considere oportuna a nuestras necesidades.

Estos métodos se implementan de forma automática, ya que JJTree es el que se encarga de generar el fichero de estado arriba mencionado (para nuestro ejemplo, **JJTejemplo1State.java** y **JJTejemplo1bisState.java**; ver Estados en JJTree para más información). A continuación se muestran algunos métodos que han sido generados automáticamente:

```
JJTejemplo1State() {
        nodes = new java.util.Stack();
        marks = new java.util.Stack();
        sp = 0;
        mk = 0;
}
boolean nodeCreated() {
        return node_created;
}
Node rootNode() {
        return (Node)nodes.elementAt(0);
}
void pushNode(Node n) {
        nodes.push(n);
        ++sp;
}
Node popNode() {
        if (--sp < mk) {
                mk = ((Integer)marks.pop()).intValue();
        }
        return (Node)nodes.pop();
}
```

5.6.2.3 Tipos de Nodos

JJTree proporciona decoraciones para dos tipos básicos de nodos, además de algunas simplificaciones sintácticas para facilitar su uso.

1. Un nodo se construye con un número definido de hijos. Recordemos que a un nodo padre se le asocian uno o más nodos hijos que son sacados de la pila e introducidos de nuevo en ella, pero formando parte de la estructura del padre. Un nodo definido se puede crear de la siguiente manera:

#ADefiniteNode (INTEGER EXPRESSION)

INTEGER EXPRESSION puede ser cualquier tipo de expresión entera. Lo más común es que dicha expresión sea una constante (de tipo entero). **ADefiniteNode** es el nombre con el cual queremos designar al nodo (no-terminal, función java) que le precede.

2. Un nodo condicional se construye con todos los nodos hijos asociados al ámbito del padre siempre que la condición del padre se evalúe a true. Si no es así, dicho nodo no se crea, devolviendo a la pila los nodos hijo que previamente habían sido asociados al nodo padre evaluado. Esto se representa de la siguiente manera:

   #ConditionalNode (BOOLEAN EXPRESSION)

   BOOLEAN EXPRESSION puede ser cualquier expresión de tipo boolean. Igual que en el caso anterior, el nombre que se le da al nodo viene dado por **ConditionalNode**. Existen dos tipos comunes de abreviaturas para dichos nodos:

   a. **#IndefiniteNode** es equivalente a **#IndefiniteNode (true)**

   b. **#GTNode (>1)** es equivalente a **#GTNode (JJTree.arity () > 1)**

   #IndefiniteNode puede llevarnos a situaciones ambiguas en JJTree cuando dicha expresión le sigue otra con paréntesis. En estos casos, se debe sustituir la expresión reducida por su versión sin reducir:

   (…..) #N (a ()) es una expresión ambigua. En su lugar, habría que utilizar

   (…..) #N (true) (a ())

   Nótese que las expresiones pasadas como parámetros a los nodos no deben tener efectos laterales. JJTree no especifica cuántas veces será evaluada la expresión.

Por defecto, JJTree trata cada no terminal como un nodo indefinido y deriva su nombre a partir de su producción. El usuario puede cambiar este nombre derivado mediante la siguiente expresión, tal y como quedó expuesto más arriba, en la zona del ejemplo:

```
void ExpSuma() #expS : {}
{
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
```

Cuando el parser reconoce al no-terminal **ExpSuma()**, comienza como nodo indefinido con nombre **expS**. Se señala en la pila, de manera que cualquier árbol parse de nodos creados y almacenados en la pila por no-terminales en la expansión de **ExpSuma()** será sacado de la misma y pasarán a formar parte de la estructura del nodo **expS** como hijos de éste. Los hijos generados tendrán como nombre en la pila **ExpMult**.

Si se quiere suprimir la creación de un nodo por una producción, se puede usar la sintaxis siguiente:

```
void ExpSuma() #void : {}
{
```

```
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
```

Ahora cualquier nodo del árbol parser almacenado en la pila que ha sido generado por el no-terminal **ExpSuma()**, permanecerá en la pila para ser sacado de la misma y convertirse en hijo de una producción ascendente en el árbol, pero no pasará a formar parte de **ExpSuma()**, ya que este nodo no está en la pila. Sus hijos (producciones) pasarán a ser los hijos del nodo inmediatamente superior a **ExpSuma()**. Esta operación se puede hacer por defecto para los nodos no decorados (sin especificación de atributos) usando la opción **NODE_DEFAULT_VOID**.

Esto se consigue añadiendo, ala zona de opciones de **ejemplo1bis.jjt**, el siguiente código:

```
options {
        MULTI=true;
        NODE_DEFAULT_VOID=true;
}
```

El efecto que tiene esta opción sobre el código es el "ahorro" de indicar cada no-terminal como **#void**:

```
void ExpSuma() : {}
{
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
```

Una forma de optimización de código utilizando el constructor # consiste en lo siguiente:

```
void ExpSuma() #void: {}
{
        (ExpMult() ( ( "+" | "-" ) ExpMult() )*)#Suma(>1)
}
```

El nodo **ExpSuma()** tendrá como hijos a uno o varios nodos llamados **Suma**. El constructor **#Suma** es de tipo condicional, e indica que es de aridad mayor que 1. Se utiliza el tipo condicional ya que no se sabe cuántos no terminales de este tipo se han de generar para reconocer la sentencia que le paso a la gramática. Ejerce de operador en forma postfija, y su ámbito se corresponde con la unidad de expansión que le precede inmediatamente.

5.6.3 Ámbitos de Nodos y Acciones de Usuario

Cada nodo está asociado a un ámbito. Para poder acceder al nodo, las acciones que el usuario defina para dicho acceso han de tener el mismo ámbito que el nodo, y además se ha de usar el método especial **jjThis** para referirse al mismo. Este identificador se declara implícitamente de la siguiente manera:

```
SimpleNode Inicio() : {}
{
        Expresion() ";"
        { return jjtThis; }
```

```
}
```

De esta forma es más fácil el acceso a los campos y métodos del nodo al que se quiere acceder. Este identificador devuelve siempre el ámbito al que pertenece el nodo.

Un ámbito es la unidad que se expande y que le precede de forma inmediata al nodo con especificación de atributos (*decorated node*); por ejemplo, puede ser una expresión entre paréntesis. Cuando la signatura de la producción se decora (puede tener al nodo por defecto de forma implícita), el ámbito es la parte derecha de la producción, incluyendo el bloque de la declaración.

También se puede usar el método **jjThis** pasándole como parámetro la parte izquierda de una expansión de referencia. Un ejemplo basado en la gramática que estoy usando es el siguiente:

```
void ExpUnaria() : {}
{
        "(" Expresion() ")" | Identificador() | <ENTERO>
}

void Identificador() #Identificador: {
        Token token;}
{
        token=<IDENTIFICADOR> { jjtThis.setName(token.image); }
}
```

Aquí, **jjtThis** "trabaja" en el ámbito del nodo en el que ha sido invocado. Es decir, las modificaciones que se hagan sólo tendrán efecto en este nodo. Esta operación se puede realizar mediante el uso de la opción **NODE_SCOPE_HOOKS**, que, por defecto, está evaluada a false (ver la sección *Node Scope Hooks* para más información). Al sacar por pantalla los nodos almacenados en la pila observamos que la salida correspondiente a la entrada **(a + b)*(c + d)** es:

```
 Inicio
 Mult
        Suma
                Identificador: a
                Identificador: b
        Suma
                Identificador: c
                Identificador: d
```

Para poder actualizar el valor del campo nombre de cada nodo, hemos de implementar nosotros mismos el método **setName()** (o cualquier otro del que queramos hacer uso, como **setFirstToken()**, **setLastToken()**, **getName()**...). La forma y el lugar correcto de implementación se explican en la zona *Node Scope Hooks* del presente documento. La acción final que realiza un usuario en el ámbito de un nodo es distinta de la de todos los demás. Cuando el código se ejecuta, los hijos del nodo son sacados de la pila y añadidos a la estructura del nodo, el cual se almacena por sí mismo en la

pila. Se puede acceder a los hijos del padre mediante el método **jjGetChild ()**. Estos métodos de acceso a los atributos de cada nodo se han implementado de forma automática, como más arriba se explica.

Otras acciones de usuario sólo pueden acceder a los hijos en la pila, debido a que aún no son parte del padre. No sirven los métodos que impliquen accesos al nodo (tales como **jjGetChild ()**).

Un nodo condicional que tenga un descriptor de nodo y sea evaluado a false, no será añadido a la pila, ni se le añadirá ningún hijo a su estructura. El usuario final, dentro del ámbito de un nodo condicional, puede precisar si el nodo ha sido creado o no mediante la llamada al método **nodeCreated** () (implementado en **SimpleNode.java**).Devuelve true si la condición del nodo se satisface, si el nodo ha sido creado y si ha sido almacenado en la pila. Devuelve false en otro caso.

5.6.4 Manejo de Excepciones

Una excepción lanzada por una expansión en el ámbito de un nodo que no es capturada en dicho ámbito, es capturada por JJTree. Cuando esto ocurre, cualquier nodo que haya sido almacenado en la pila es eliminado junto con su ámbito correspondiente. Entonces, la excepción se vuelve a recoger.

Este tipo de manejo de las excepciones tiene por objetivo hacer posible una implementación de recuperación de errores y continuar con la generación del parser a partir de un estado conocido.

Es importante hacer notar que, actualmente, JJTree no detecta cuándo una excepción es lanzada a partir de una acción de usuario dentro del ámbito de un nodo, con lo que, probablemente, una excepción puede ser tratada de forma incorrecta.

5.6.5 *Node Scope Hooks*

Esta opción altera el comportamiento de JJTree, de manera que, automáticamente, todos los tokens reconocidos son capturados. Si la opción **NODE_SCOPE_HOOK** está evaluada a true, JJTree genera llamadas a dos métodos definidos por el usuario en la entrada y salida de cada ámbito de nodo. Dichos métodos son:

```
void jjtreeOpenNodeScope(Node nodo)
void jjtreeCloseNodeScope(Node nodo)
```

Estos métodos han de ser definidos de manera que tengan acceso al primer y último token del nodo generado. De esta manera, se puede iterar desde el primer token encontrado hasta el último, sin miedo a que cambie el ámbito de la iteración, debido a que dichos métodos permanecen en el ámbito del nodo actual.

Se han de añadir los campos siguientes: **Token primero**, **ultimo**;

Se han de implementar en el fichero generado por javacc de forma automática llamado **SimpleNode.jav**a, ya que de esta clase, javac se alimenta de todos los

métodos usados en el fuente con extensión **.jjt**.

El fichero **SimpleNode.java** del **ejemplo3.jjt** se presenta a continuación como apoyo a la explicación dada anteriormente:

```
public class SimpleNode implements Node {
        protected Node parent;
        protected Node[] children;
        protected int id;
        protected String name;
        protected Token token,first,last;
        protected ejemplo3 parser;

        public SimpleNode(int i) {
                id = i;
        }
        public SimpleNode(ejemplo3 p, int i) {
                this(i);
                parser = p;
        }

        public void jjtOpen() {
        }
        public void jjtClose() {
        }
        public void jjtSetParent(Node n) { parent = n; }
        public Node jjtGetParent() { return parent; }
        public void jjtAddChild(Node n, int i) {
                if (children == null) {
                        children = new Node[i + 1];
                } else if (i >= children.length) {
                        Node c[] = new Node[i + 1];
                        System.arraycopy(children, 0, c, 0, children.length);
                        children = c;
                }
                children[i] = n;
        }
        public Node jjtGetChild(int i) {
                return children[i];
        }
        public int jjtGetNumChildren() {
                return (children == null) ? 0 : children.length;
        }
        /* You can override these two methods in subclasses of SimpleNode to
        customize the way the node appears when the tree is dumped.  If
        your output uses more than one line you should override
        toString(String), otherwise overriding toString() is probably all
        you need to do. */
        public void setName(String s) {
                this.name=s;
```

```
        }
        public String getName() {
                return this.name;
        }
        public void setFirstToken(Token t) {
                this.first=t;
        }
        public Token getFirstToken() {
                return this.first;
        }
        public void setLastToken(Token t) {
                this.last=t;
        }
        public Token getLastToken() {
                return this.last;
        }
        public String toString() {
                return ejemplo3TreeConstants.jjtNodeName[id]+":"+name;
         }
         public String toString(String prefix) { return prefix + toString();
         }
        /* Override this method if you want to customize how the node dumps
        out its children. */
        public void dump(String prefix) {
                System.out.println(toString(prefix));
                if (children != null) {
                        for (int i = 0; i < children.length; ++i) {
                                SimpleNode n = (SimpleNode)children[i];
                                if (n != null) {
                                        n.dump(prefix + " ");
                                }
                        }
                }
        }
}
```

Aquellas variables y métodos que se presentan en negrita son los que necesitamos definir (o redefinir) para que javac reconozca todos aquellos métodos que hemos definido en el fichero principal .jjt

El resto de los métodos son generados automáticamente por javacc, pudiendo modificarlos siempre y cuando seamos conscientes de las modificaciones llevadas a cabo. Cualquier acceso a un nodo o cualquier operación incorrecta llevará a la generación de errores. Si no los modificamos, tenemos la seguridad de que dichos métodos realizan las acciones que les son asignadas por defecto.Otro uso que se le puede dar es almacenar el parser objeto en el nodo, de manera que se puede proporcionar el estado que debe ser compartido por todos los nodos generados por el parser. Por ejemplo, el parser podría mantener una tabla de símbolos.

5.6.6 El ciclo de vida de un nodo

Un nodo pasa por una secuencia determinada de pasos en su generación. La secuencia vista desde el propio nodo es la siguiente:

1. El constructor del nodo se llama con un único parámetro de tipo entero. Dicho parámetro nos dice qué tipo de nodo es, siendo esto especialmente útil en modo simple. JJTree genera automáticamente un fichero llamado parserTreeConstants.java y declara constantes en Java para el identificador del nodo. Además, declara un array de Strings llamado jjtNodeName [], el cual asigna identificadores enteros a los nombres de los nodos. Para nuestro ejemplo2, el fichero es el siguiente:

```
public interface ejemplo2TreeConstants{
    public int JJTINICIO = 0;
    public int JJTVOID = 1;
    public int JJTSUMA = 2;
    public int JJTMULT = 3;
    public int JJTIDENTIFICADOR = 4;
    public int JJTENTERO = 5;
    public String[] jjtNodeName = {
        "Inicio",
        "void",
        "Suma",
        "Mult",
        "Identificador",
        "Entero",
      };
    }
```

2. Se realiza una llamada al método jjtOpen ().

3. Si la opción **NODE_SCOPE_HOOK** está activa, se llama al método parser definido por el usuario **openNodeScope ()** y se le pasa el nodo como parámetro. Por ejemplo, podría almacenar el primer token en el nodo:

```
static void jjtreeOpenNodeScope(Node nodo) {
        ((SimpleNode)nodo).setFirstToken(getToken(1));
}
```

   Este método ha de implementarse en la clase principal, es decir, en el **ejemplo2.jjt**

4. Si una excepción sin manejador se lanza mientras que el nodo se está generando, dicho nodo se descarta. JJTree nunca volverá a hacer referencia a este nodo. Dicho nodo no se cerrará, y en el método de usuario **closeNodeHook()** no utilizará dicho nodo como parámetro en una llamada.

5. En otro caso, si el nodo es condicional, y su condición se evalúa a false,

el nodo se descarta. No se cerrará dicho nodo, a no ser que el método definido por el usuario **closeNodeHook ()** pueda utilizarse en una llamada con este nodo como parámetro.

6. En otro caso, el número de hijos del nodo especificado en la expresión entera del nodo definido, o todos los nodos que fueron almacenados en la pila en un ámbito de nodo condicional, son añadidos al nodo (dicho número es la aridad del nodo; si es condicional, se define **#Id(>1))**. El orden en el que se añaden no se especifica.
7. Se llama al método **jjtClose ()**.
8. El nodo es almacenado en la pila.
9. Si la opción **NODE_SCOPE_HOOK** está activa, el método parser definido por el usuario **closeNodeScope ()** se llama y utiliza el nodo como parámetro.
10. Si el nodo no es el nodo raíz, se añade como estructura de hijo a otro nodo y se llama a su método **jjtParent ()**.

5.6.7 *Visitor Support*

JJTree provee algunos soportes básicos para el patrón de diseño visitor. Si la opción **VISITOR** está a true, JJTree añadirá un método **jjtAccept ()** en todas las clases de nodo que genere, además de la creación de una interfaz de usuario que pueda ser implementada y pasada a los nodos.

El nombre de dicha interfaz se construye mediante la adición **Visitor** al nombre del parser. La interfaz se genera cada vez que JJTree se ejecuta, de manera que representa de forma exacta el conjunto de nodos usados por el parser. Si la clase implementada no ha sido actualizada para los nodos nuevos generados, producirá errores en tiempo de compilación.

5.6.7.1 Parámetros de Ejecución

JJTree proporciona opciones para la línea de comando y para las opciones de declaraciones de JavaCC:

1. **BUILD_NODE_FILES** (**true** por defecto): genera implementaciones de muestra para SimpleNode y para cualesquiera otros nodos usados en la gramática.
2. **MULTI** (**false** por defecto): genera el árbol parse en modo multi. Cada nodo (no-terminal) creado y no declarado como **void**, genera un fichero de nombre **nombreNoTerminal.java**, en el cual cada nodo extiende de **SimpleNode**. Estos ficheros se generan de forma automática.
3. **NODE_DEFAULT_VOID** (**false** por defecto): en vez de crear cada producción no-decorada como un nodo indefinido, lo crea void en su

lugar. Cualquier nodo decorado no se verá afectado por esta opción.

4. **NODE_FACTORY** (**false** por defecto): usa un método de generación para construir nodos con la siguiente estructura:
   public static Node jjtCreate (int id)
5. **NODE_PACKAGE** (**""** por defecto): el paquete para generar el nodo en una clase. Por defecto, contiene el paquete del parser.
6. **NODE_PREFIX** (**"AST"** por defecto): el prefijo usado para construir los nombres de clase del nodo a partir de los identificadores del nodo en modo multi. Es el prefijo que se pone automáticamente por defecto cuando se generan los ficheros que contienen las implementaciones de los nodos que extienden de **SimpleNode**.
7. **NODE_SCOPE_HOOK** (**false** por defecto): inserta llamadas a los métodos parser definidos por el usuario en cada entrada y salida de cada ámbito de nodo. Ver Node Scope Hooks
8. **NODE_USERS_PARSER** (**false** por defecto): JJTree usará una forma alternativa de las rutinas de construcción de nodos donde se le pasa el parser del objeto. Por ejemplo:
   public static Node MiNodo.jjtCreate(MiParser p, int id);
   MiNodo (MiParser p, int id);
9. **STATIC** (**true** por defecto): genera código para un parser static. Debe ser usado de forma consecuente con las opciones equivalentes proporcionadas por JavaCC. El valor de esta opción se muestra en el fuente de JavaCC.
10. **VISITOR** (**false** por defecto): inserta un método jjtAccept () en las clases del nodo, y genera una implementación de visitor con una entrada para cada tipo de nodo usado en la gramática.
11. **VISITOR_EXCEPTION** (**""** por defecto): si esta opción está activa, se usa en las estructuras de los métodos **jjtAccept ()** y **visit ()**. Mucho cuidado porque esta opción se eliminará en versiones posteriores de JJTree.

5.6.8 Estados en JTree

JJTree mantiene su estado en un campo de la clase del parser llamado jjtree. Se puede usar métodos en este miembro para manipular la pila del nodo. Algunos de los métodos generados automáticamente para esta clase están expuestos más arriba. También se indica en qué fichero se encuentra almacenado el código generado.

5.6.8.1 Objetos Node

Todos los nodos AST (*Abstract Syntax Tree*) deben implementar esta interfaz. Proporciona un motor básico para construir relaciones padre-hijos entre nodos.

Esta interfaz es común para todos los archivos **.jjt** que se lancen mediante el ejecutable **jjtree**. El contenido es:

```
public interface Node {
        /* Este método se llama después de que el nodo haya pasado a ser el nodo
        actual. Indica que se le pueden añadir los nodos hijo*/
        public void jjtOpen ();

        /* Este método se llama después de haber añadido todos los nodos hijo*/
        public void jjtClose ();

        /* Los siguientes 2 métodos se usan para asignar/devolver el padre de un
        nodo*/
        public void jjtSetParent (Node n);
        public Node jjtGetParent () ;

        /*Este método añade su argumento a la lista de nodos hijo*/
        public void jjtAddChild (Node n, int i);

        /*Este método devuelve un nodo hijo. Se numeran desde 0, de izquierda a
        derecha*/
        public Node jjtGetChild (int i);

        /*Devuelve el número de hijos que tiene el nodo*/
        int jjtGetNumChildren ();
}
```

La clase **SimpleNode** implementa la interfaz **Node**, y si no existe, se genera de forma automática por JJTree. Se puede usar esta clase como una plantilla, como una superclase para las implementaciones de nodos, o se puede modificar a conveniencia.

SimpleNode provee, de forma adicional, un mecanismo simple de "recolector" recursivo del nodo y sus hijos. Dicho método es el que aparece en los ejemplos arriba descritos como **dump()**.

El parámetro de tipo **String** del método **dump()** se usa como prefijo para indicar la estructura jerárquica del árbol.

Si la opción **VISITOR** está activa, se genera otro método que puede ser de utilidad:

```
public void childrenAccept (MyParserVisitor visitor);
```

Este método recorre los nodos hijos en orden, invitándoles a aceptar a visitor. Este método es muy útil cuando se ha implementado una estructura en preorden o postorden.

5.6.8.2 Pasos de Ejecución

La ejecución de los ficheros **.bat** han de lanzarse en el editor de comandos de MS-DOS.

1. Si el fichero a compilar tiene extensión **.jj**:

```
javacc archivo.jj
```

```
javac archivo.java
java archivo
```

2. Si el fichero tiene extensión **.jjt**:

```
jjtree archivo.jjt
javacc archivo.jj
javac archivo.java
java archivo
```

Los archivos generados al ejecutar **jjtree <archivo.jjt>** son:

archivoTreeConstants.java
JJTarchivoState.java
Node.java
SimpleNode.java

Si el fuente está en modo multi, se generará un fichero **.java** por cada no-terminal no declarado como **void**, es decir, que esté decorado. Los nombres de los ficheros serán los mismos que los nombres de los no-terminales.

Los archivos generados al ejecutar **javacc <archivo.jj>** son:

archivo.java
archivoConstants.java
archivoTokenManager.java
TokenMgrError.java
ParseException.java
Token.java
SimpleCharStream.java

El siguiente paso es la modificación del fichero **SimpleNode.java** con los métodos que nos sean de utilidad, esto es, que hemos invocado en el fuente **.jjt**.

De esta manera, no se generarán errores tipo "*method not found*" o "*symbol not found*" a la hora de compilar el fichero **.java** generado automáticamente por javacc.

Hemos de ejecutar el comando javac en el directorio contenedor o, si queremos ahorrarnos tiempo, actualizar el *path* con la dirección en la que se encuentra **javac.bat** y **java.bat**

La linea de comandos a ejecutar, en el caso de encontrarnos en el director contenedor de la javac es:

```
C:\jdk1.5.0_01\bin> javac -classpath <camino de los ficheros fuente> ejemplo.java
C:\jdk1.5.0_01\bin> java -classpath <camino de los ficheros fuente> ejemplo
```

Nos pedirá la introducción de la sentencia(s) a reconocer, si la entrada seleccionada es la estándar, o pasarle el fichero en el que se encuentra el código a reconocer mediante el comando

```
C:\jdk1.5.0_01\bin> java -classpath <path de los ficheros fuente> ejemplo > fichero
```

5.6.9 Manual de Referencia de JJDoc

JJDoc toma la especificación del parser de JavaCC y produce la correspondiente documentación para la gramática BNF. Puede funcionar de tres formas distintas, dependiendo del comando introducido en la zona de opciones:

- **TEXT**: Por defecto, esta opción está a false.Inicializando esta opción a true, JJDoc genera una descripción simple en formato texto de la gramática en BNF. Algunos formateos se hacen vía caracteres tabuladores, pero el objetivo es dejarlo tan simple como sea posible.
- **ONE_TABLE**: Por defecto, true.El valor por defecto de ONE_TABLE se usa para generar una única tabla HTML para la notación BNF. Actualizar el valor a false supone la generación de una tabla por cada producción gramatical.
- **OUTPUT_FILE**: El funcionamiento por defecto es poner la salida de JJDoc en un fichero junto con la extensión **.html** o **.txt** añadida como sufijo al nombre del fichero de entrada. Puede suministrar un nombre de fichero distinto con esta opción.

Cuando está activada la opción **TEXT** o está puesta a false la opción **ONE_TABLE**, los comentarios en el fuente JavaCC que preceden inmediatamente a una producción son obviados por la documentación generada.

Para **ONE_TABLE**, obtenemos un fichero **ejemplo2.html** cuyo contenido es:

BNF for ejemplo2.jj (NON-TERMINALS)

Inicio	::=	Expresion ";"
Expresion	::=	ExpSuma
ExpSuma	::=	(ExpMult (("+" | "-") ExpMult)*)
ExpMult	::=	(ExpUnaria (("*" | "/" | "%") ExpUnaria)*)
ExpUnaria	::=	"(" Expresion ")"
	|	Identificador
	|	Entero
Identificador	::=	(<IDENTIFICADOR>)
Entero	::=	<ENTERO>

Para **TEXT=true**, obtenemos un fichero **ejemplo2.txt** cuyo contenido es:

```
DOCUMENT START
NON-TERMINALS
        Inicio    :=      Expresion ";"
        Expresion         :=      ExpSuma
        ExpSuma           :=      ( ExpMult ( ( "+" | "-" ) ExpMult )* )
        ExpMult :=        ( ExpUnaria ( ( "*" | "/" | "%" ) ExpUnaria )* )
```

```
        ExpUnaria       :=      "(" Expresion ")"
                |       Identificador
                |       Entero
        Identificador   :=      ( <IDENTIFICADOR> )
        Entero  :=      <ENTERO>
DOCUMENT END
```

Capítulo 6

Tabla de símbolos

6.1 Visión general

También llamada «tabla de nombres» o «tabla de identificadores», se trata sencillamente de una estructura de datos de alto rendimiento que almacena toda la información necesaria sobre los identificadores de usuario. Tiene dos funciones principales:

- Efectuar chequeos semánticos.
- Generar código.

Además, esta estructura permanece en memoria sólo en tiempo de compilación, no de ejecución, excepto en aquellos casos en que se compila con opciones de depuración. Los intérpretes también suelen mantener la tabla de símbolos en memoria durante la ejecución, ya que ésta se produce simultáneamente con la traducción.

La tabla de símbolos almacena la información que en cada momento se necesita sobre las variables del programa; información tal como: nombre, tipo, dirección de localización en memoria, tamaño, etc. Una adecuada y eficaz gestión de la tabla de símbolos es muy importante, ya que su manipulación consume gran parte del tiempo de compilación. De ahí que su eficiencia sea crítica.

La tabla de símbolos también sirve para guardar información referente a los tipos de datos creados por el usuario, los tipos enumerados y, en general, cualquier identificador creado por el usuario. En estos casos, el desarrollador puede optar por mezclar las distintas clases de identificadores en una sola tabla, o bien disponer de varias tablas, donde cada una de ellas albergará una clase distinta de identificadores: tabla de variables, tabla de tipos de datos, tabla de funciones de usuario, etc. En lo que sigue nos vamos a centrar principalmente en las variables de usuario.

6.2 Información sobre los identificadores de usuario

La información que el desarrollador decida almacenar en esta tabla dependerá de las características concretas del traductor que esté desarrollando. Entre esta información puede incluirse:

- Nombre del elemento. El nombre o identificador puede almacenarse limitando o no la longitud del mismo. Si se almacena con límite empleando un tamaño máximo fijo para cada nombre, se puede aumentar la velocidad de creación y manipulación, pero a costa de limitar la longitud de los nombres en unos

casos y desperdiciar espacio en la mayoría. El método alternativo consiste en habilitar la memoria que necesitemos en cada caso para guardar el nombre. En C esto es fácil con el tipo char *; si hacemos el compilador en Modula-2, por ejemplo, habría que usar el tipo ADDRESS. En el caso de Java, esto no entraña dificultad alguna gracias al tipo String. Las búsquedas en la tabla de símbolos suelen hacerse por este nombre; por ello, y para agilizar al máximo esta operación, la tabla de símbolos suele ser una tabla de dispersión (*hash*) en la que las operaciones de búsqueda e inserción poseen un coste aproximadamente constante ($\approx O(1)$).

- Tipo del elemento. Cuando se almacenan variables, resulta fundamental conocer el tipo de datos a que pertenece cada una de ellas, tanto si es primitivo como si no, con objeto de poder controlar que el uso que se hace de tales variables es coherente con el tipo con que fueron declaradas. El capítulo 7 está exclusivamente dedicado a esta gestión.
- Dirección de memoria en que se almacenará su valor en tiempo de ejecución. Esta dirección es necesaria, porque las instrucciones que referencian a una variable deben saber donde encontrar el valor de esa variable en tiempo de ejecución con objeto de poder generar código máquina, tanto si se trata de variables globales como de locales. En lenguajes que no permiten la recursión, las direcciones se van asignando secuencialmente a medida que se hacen las declaraciones. En lenguajes con estructuras de bloques, la dirección se da con respecto al comienzo del área de memoria asignada a ese bloque (función o procedimiento) en concreto.
- Valor del elemento. Cuando se trabaja con intérpretes sencillos, y dado que en un intérprete se solapan los tiempos de compilación y ejecución, puede resultar más fácil gestionar las variables si almacenamos sus valores en la tabla de símbolos. En un compilador, no obstante, la tabla de símbolos no almacena nunca el valor.
- Número de dimensiones. Si la variable a almacenar es un array, también pueden almacenarse sus dimensiones. Aunque esta información puede extraerse de la estructura de tipos, se puede indicar explícitamente para un control más eficiente.
- Tipos de los parámetros formales. Si el identificador a almacenar pertenece a una función o procedimiento, es necesario almacenar los tipos de los parámetros formales para controlar que toda invocación a esta función sea hecha con parámetros reales coherentes. El tipo de retorno también se almacena como tipo del elemento.
- Otra información. Con objeto de obtener resúmenes estadísticos e información varia, puede resultar interesante almacenar otros datos: números de línea en los que se ha usado un identificador, número de línea en que se declaró, tamaño del registro de activación (ver apartado 9.3.2), si es una variable

global o local, en qué función fue declarada si es local, etc.

6.3 Consideraciones sobre la tabla de símbolos

La tabla de símbolos puede inicilizarse con cierta información útil, que puede almacenarse en una única estructura o en varias:

Constantes: PI, E, NUMERO_AVOGADRO, etc.

Funciones de librería: EXP, LOG, SQRT, etc.

Palabras reservadas. Algunos analizadores lexicográficos no reconocen directamente las palabras reservadas, sino que sólo reconocen identificadores de usuario. Una vez tomado uno de la entrada, lo buscan en la tabla de palabras reservadas por si coincide con alguna; si se encuentra, devuelven al analizador sintáctico el *token* asociado en la tabla; si no, lo devuelven como identificador de verdad. Esto facilita el trabajo al lexicográfico, es más, dado que esta tabla es invariable, puede almacenarse en una tabla de dispersión perfecta (aquélla en la que todas las búsquedas son exactamente de O(1)) constituyendo así una alternativa más eficiente a PCLex o JavaCC tal y como lo hemos visto.

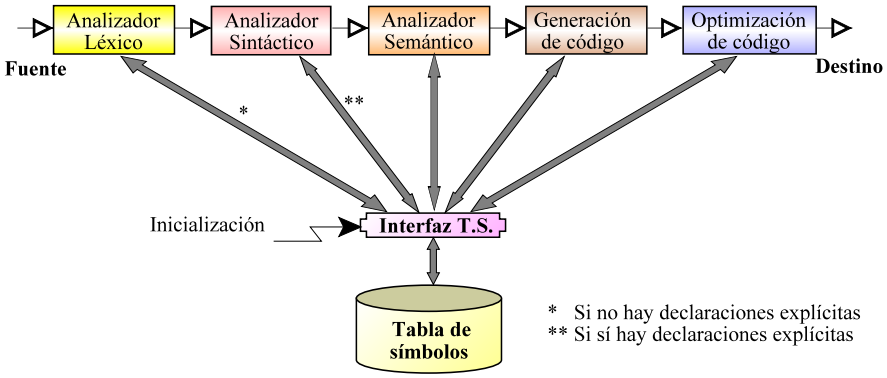


Figura 6.1. Accesos a la tabla de símbolos por parte de las distintas fases de un compilador

En general, conforme avanza la etapa de análisis y van apareciendo nuevas declaraciones de identificadores, el analizador léxico o el analizador sintáctico según la estrategia que sigamos, insertará nuevas entradas en la tabla de símbolos, evitando siempre la existencia de entradas repetidas. El analizador semántico efectúa las comprobaciones sensibles al contexto gracias a la tabla de símbolos y, ya en la etapa de síntesis, el generador de código intermedio usa las direcciones de memoria asociadas a cada identificador para generar un programa equivalente al de entrada. Por regla general el optimizador de código no necesita hacer uso de ella, aunque nada impide que pueda accederla.

Aunque la eficiencia en los accesos y manipulación de la tabla de símbolos

son primordiales para hacer un buen compilador, suele ser conveniente realizar un desarrollo progresivo del mismo. Así, en las fases tempranas de la construcción del compilador, dada su complejidad es mejor incluir una tabla de símbolos lo más sencilla posible (por ejemplo, basada en una lista simplemente encadenada) para evitar errores que se difundan por el resto de fases del compilador. Una vez que las distintas fases funcionen correctamente puede concentrarse todo el esfuerzo en optimizar la gestión de la tabla de símbolos. Para que este proceso no influya en las fases ya desarrolladas del compilador es necesario dotar a la tabla de símbolos de una interfaz claramente definida e invariable desde el principio: en la optimización de la tabla de símbolos se modifica su implementación, pero se mantiene su interfaz. Todo este proceso puede verse en la figura 6.1.

En esta figura se menciona una distinción importante entre la construcción de compiladores para lenguajes que obligan al programador a declarar todas las variables (C, Modula-2, Pascal, Java, etc.), y lenguajes en los que las variables se pueden usar directamente sin necesidad de declararlas (BASIC, Logo, etc.). Cuando un lenguaje posee área de declaraciones y área de sentencias, la aparición de un identificador tiene una semántica bien diferente según el área en que se encuentre:

- Si aparece en el área de declaraciones, entonces hay que insertar el identificador en la tabla de símbolos, junto con la información que de él se conozca. Si el identificador ya se encontraba en la tabla, entonces se ha producido una redeclaración y debe emitirse un mensaje de error semántico.
- Si aparece en el área de sentencias, entonces hay que buscar el identificador en la tabla de símbolos para controlar que el uso que de él se está haciendo sea coherente con su tipo. Si el identificador no se encuentra en la tabla, entonces es que no ha sido previamente declarado, lo que suele considerarse un error semántico del que hay que informar al programador.

En estos casos, el analizador lexicográfico, cuando se encuentra un identificador desconoce en qué área lo ha encontrado, por lo que no resulta fácil incluir una acción léxica que discrimine entre realizar una inserción o una búsqueda; así, deberá ser el analizador sintáctico quien haga estas operaciones. Así, el atributo del *token* identificador suele ser su propio nombre.

Sin embargo, hay unos pocos lenguajes de programación en los que no es necesario declarar las variables, bien porque sólo existe un único tipo de datos (como en nuestro ejemplo de la calculadora donde todo se consideran valores enteros) o bien porque el propio nombre de la variable sirve para discernir su tipo (en BASIC las variables que acaban en "$" son de tipo cadena de caracteres). En estos casos, el propio analizador lexicográfico puede insertar el identificador en la tabla de símbolos la primera vez que aparezca; antes de insertarlo realiza una operación de búsqueda por si ya existía. Y sea cual sea el caso, se obtiene la entrada o índice de la tabla en la que se encontraba o se ha incluido; dicho índice será el valor que el analizador léxico pasa al sintáctico como atributo de cada identificador.

Por último, es importante recalcar que la tabla de símbolos contiene información útil para poder compilar, y por tanto sólo existe en tiempo de compilación, y no de ejecución. Sin embargo, en un intérprete dado que la compilación y la ejecución se producen a la vez, la tabla de símbolos permanece en memoria todo el tiempo.

6.4 Ejemplo: una calculadora con variables

Llegados a este punto, vamos a aumentar las posibilidades de la calculadora propuesta en epígrafes anteriores, incluyendo la posibilidad de manipular variables. Para simplificar el diseño no incluiremos un área de declaraciones de variables, sino que éstas se podrán utilizar directamente inicializándose a 0, puesto que tan sólo disponemos del tipo entero.

Para introducir las variables en la calculadora necesitamos una nueva construcción sintáctica con la que poder darle valores: se trata de la asignación. Además, continuaremos con la sentencia de evaluación y visualización de expresiones a la que antepondremos la palabra reservada **PRINT**. En cualquier expresión podrá intervenir una variable que se evaluará al valor que tenga en ese momento. Así, ante la entrada:

❶ a := 7 * 3;
❷ b := 3 * a;
❸ a := a + b;

se desea almacenar en la variable **a** el valor 21, en la **b** el valor 63 y luego modificar **a** para que contenga el valor 84. Para conseguir nuestros propósitos utilizaremos una tabla de símbolos en la que almacenaremos tan sólo el nombre de cada variable, así como su valor ya que estamos tratando con un intérprete. La tabla de símbolos tendrá

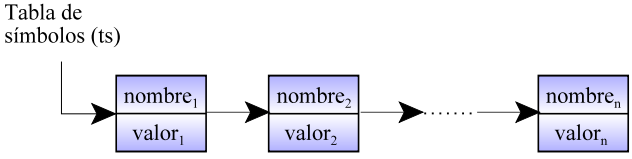


Figura 6.2. Una de las tablas de símbolos más simples (y más ineficientes) que se pueden construir

una estructura de lista no ordenada simplemente encadenada (ver figura 6.2), ya que la claridad en nuestro desarrollo prima sobre la eficiencia. En los desarrollos con Java haremos uso de la clase **HashMap**.

La figura 6.3 muestra cómo se debe actualizar la tabla de símbolos para reflejar las asignaciones propuestas. Así, en la sentencia ❶ una vez evaluada la expresión 7*3 al valor 21 se inspeccionará la tabla de símbolos en busca del identificador **a**. Dado que éste no está, el analizador léxico lo creará inicializándolo a 0; a continuación la regla asociada a la asignación modificará la entrada de la **a** para

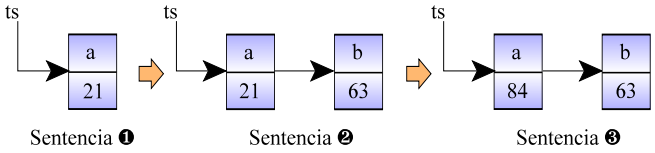


Figura 6.3. Situación de la tabla de símbolos tras ejecutar algunas sentencias de asignación

asignarle el valor 21. En la sentencia ❷, la evaluación de la expresión requiere buscar **a** en la tabla de símbolos para recoger su valor -21- para multiplicarlo por 3. El resultado -63- se almacena en la entrada asociada a la **b** de igual forma que se hizo para la variable **a** en la sentencia ❶. Por último, la sentencia ❸ evalúa una nueva expresión en la que intervienen **a** y **b**, asignando su suma -84- a la entrada de **a** en la tabla de símbolos.

6.4.1 Interfaz de la tabla de símbolos

Como ya se ha comentado, la interfaz de la tabla de símbolos debe quedar clara desde el principio de manera que cualquier modificación en la implementación de la tabla de símbolos no tenga repercusión en las fases del compilador ya desarrolladas. Las operaciones básicas que debe poseer son:

- **crear()**: crea una tabla vacía.
- **insertar(simbolo)**: añade a la tabla el símbolo dado.
- **buscar(nombre)**: devuelve el símbolo cuyo nombre coincide con el parámetro. Si el símbolo no existe devuelve null.
- **imprimir()**: a efectos informativos, visualiza por la salida estándar la lista de variables almacenadas en la tabla de símbolos junto con sus valores asociados.

6.4.2 Solución con Lex/Yacc

Asumimos que no hay área de declaraciones en nuestro lenguaje, por lo que la inserción de los símbolos en la tabla se debe hacer desde una acción léxica. Como atributo del *token* ID se usará un puntero a su entrada en la tabla de símbolos.

Las acciones semánticas usarán este puntero para acceder al valor de cada variable: si el identificador está a la izquierda del *token* de asignación, entonces se machacará el valor; y si forma parte de una expresión, ésta se evaluará al valor de la variable.

Por otro lado, el atributo del *token* NUM sigue siendo de tipo **int**. Dado que se necesitan atributos de tipos distintos (para NUM y para ID), habra que declarar un **%union** en Yacc, de la forma:

```
%union {
```

```
        int numero;
        simbolo * ptrSimbolo;
}
```

y los terminales y no terminales de la forma:

```
%token <numero> NUM
%token <ptrSimbolo> ID
%type <numero> expr
```

Para finalizar, podemos optar por permitir asignaciones múltiples o no. Un ejemplo de asignación múltiple es:

```
a := b := c := 16;
```

que asigna el valor 16 a las variables **c**, **b** y **a** en este orden. Para hacer esto, consideraremos que las asignaciones vienen dadas por las siguientes reglas:

```
asig    :       ID ASIG expr
        |       ID ASIG asig
        ;
```

La figura 6.4 ilustra el árbol sintáctico que reconoce este ejemplo. Como puede verse en él, el no terminal **asig** también debe tener asociado como atributo el campo **numero** del **%union**, con objeto de ir propagando el valor de la expresión y poder realizar las asignaciones a medida que se asciende en el árbol sintáctico.

La tabla de símbolos viene dada por el código siguiente, almacenado en el fichero **TabSimb.c**:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  typedef struct _simbolo {
4          struct _simbolo * sig;
5          char nombre [20];
6          int valor;
```

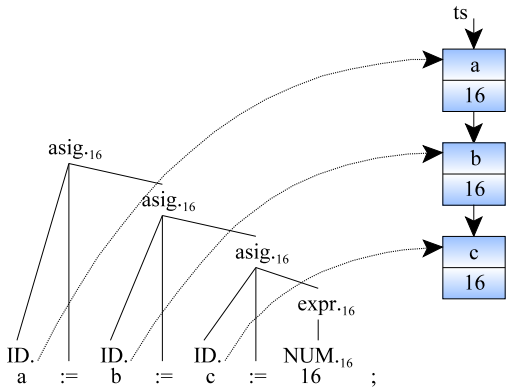


Figura 6.4. Árbol sintáctico para reconocer una asignación múltiple

```
7
```

```
} simbolo;
simbolo * crear() {
        return NULL;
};
void insertar(simbolo **pT, simbolo *s) {
        s->sig = (*pT);
        (*pT) = s;
};
simbolo * buscar(simbolo * t, char nombre[20]){
        while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
                t = t->sig;
        return (t);
};
void imprimir(simbolo * t) {
        while (t != NULL) {
                printf("%s\n", t->nombre);
                t = t->sig;
        }
};
```

El programa Lex resulta sumamente sencillo. Tan sólo debe reconocer los lexemas e insertar en la tabla de símbolos la primera vez que aparezca cada identificador. El siguiente fichero **Calcul.lex** ilustra cómo hacerlo:

```
%%
[0-9]+          {
                    yylval.numero = atoi(yytext);
                    return NUMERO;
                }
":="            { return ASIG; }
"PRINT"         { return PRINT; }
[a-zA-Z][a-zA-Z0-9]*   {
                        yylval.ptrSimbolo = buscar(t,yytext);
                        if (yylval.ptrSimbolo == NULL) {
                            yylval.ptrSimbolo=(simbolo *) malloc(sizeof(simbolo));
                            strcpy(yylval.ptrSimbolo->nombre, yytext);
                            yylval.ptrSimbolo->valor=0;
                            insertar(&t, yylval.ptrSimbolo);
                        }
                        return ID;
                    }
[ \t\n]+            { ; }
.                   { return yytext[0]; }
```

El programa Yacc tan sólo difiere de la calculadora tradicional en los accesos a la tabla de símbolos para obtener y alterar el valor de cada variable. El siguiente código se supone almacenado en el fichero **Calcuy.yac**:

```
%{
#include "TabSimb.c"
simbolo * t;
```

```
%}
%union {
        int numero;
        simbolo * ptrSimbolo;
}
%token <numero> NUMERO
%token <ptrSimbolo> ID
%token  ASIG PRINT
%type <numero> expr asig
%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO
%%
prog    :       prog asig  ';'          { printf("Asignacion(es) efectuada(s).\n"); }
        |       prog PRINT expr  ';'    { printf("%d\n",$3); }
        |       prog error ';'          { yyerrok; }
        |       /* Épsilon */
        ;
asig    :       ID ASIG expr    {
                                        $$ = $3;
                                        $1->valor = $3;
                                }
        |       ID ASIG asig    {
                                        $$ = $3;
                                        $1->valor = $3;
                                }
        ;
expr    :       expr '+' expr       {$$ = $1 + $3;}
        |       expr '-' expr       {$$ = $1 - $3;}
        |       expr '*' expr       {$$ = $1 * $3;}
        |       expr '/' expr       {$$ = $1 / $3;}
        |       '(' expr ')'        {$$ = $2;}
        |       '-' expr    %prec MENOS_UNARIO  {$$ = - $2;}
        |       ID                  {$$ = $1->valor; }
        |       NUMERO              {$$ = $1;}
        ;
%%
#include "Calcul.c"
#include "errorlib.c"
void main()
{       t = crear();
        yyparse ();
        imprimir(t);
}
```

6.4.3 Solución con JFlex/Cup

Dado que Java incorpora el tipo **HashMap** de manera predefinida, nuestra tabla de símbolos será una clase **Tablasimbolos** que encapsula en su interior una tabla

de dispersión y proporciona al exterior tan sólo las operaciones del punto 6.4.1. Como clave de la tabla usaremos un **String** (el nombre del símbolo) y como dato utilizaremos un objeto de tipo **Simbolo** que, a su vez, contiene el nombre y el valor de cada variable. El fichero **Simbolo.java** es:

```
class Simbolo{
	String nombre;
	Integer valor;
	public Simbolo(String nombre, Integer valor){
		this.nombre = nombre;
		this.valor = valor;
	}
}
```

Y **TablaSimbolos.java** quedaría:

```
import java.util.*;
public class TablaSimbolos{
	HashMap t;
	public TablaSimbolos(){
		t = new HashMap();
	}
	public Simbolo insertar(String nombre){
		Simbolo s = new Simbolo(nombre, new Integer(0));
		t.put(nombre, s);
		return s;
	}
	public Simbolo buscar(String nombre){
		return (Simbolo)(t.get(nombre));
	}
	public void imprimir(){
		Iterator it = t.values().iterator();
		while(it.hasNext()){
			Simbolo s = (Simbolo)it.next();
			System.out.println(s.nombre + ": "+ s.valor);
		}
	}
}
```

La tabla de símbolos debería ser vista tanto por el analizador sintáctico como por el léxico, con objeto de poder acceder a sus elementos. Aunque en este ejemplo no es estrictamente necesario, haremos que la tabla de símbolos sea creada por el programa principal como dato propio del analizador sintáctico (ya que estamos en un análisis dirigido por sintaxis) y se pasará al analizador léxico como un parámetro más en el momento de su construcción. De esta manera el fichero **Calcul.jflex** quedaría:

```
import java_cup.runtime.*;
import java.io.*;
%%
%{
	private TablaSimbolos tabla;
	public Yylex(Reader in, TablaSimbolos t){
```

```
7                   this(in);
8                   this.tabla = t;
9               }
10  %}
11  %unicode
12  %cup
13  %line
14  %column
15  %%
16  "+"         { return new Symbol(sym.MAS); }
17  "-"         { return new Symbol(sym.MENOS); }
18  "*"         { return new Symbol(sym.POR); }
19  "/"         { return new Symbol(sym.ENTRE); }
20  ";"         { return new Symbol(sym.PUNTOYCOMA); }
21  "("         { return new Symbol(sym.LPAREN); }
22  ")"         { return new Symbol(sym.RPAREN); }
23  ":="        { return new Symbol(sym.ASIG); }
24  "PRINT"             { return new Symbol(sym.PRINT); }
25  [:jletter:][:jletterdigit:]*        {
26                                          Simbolo s;
27                                          if ((s = tabla.buscar(yytext())) == null)
28                                              s = tabla.insertar(yytext());
29                                          return new Symbol(sym.ID, s);
30                                      }
31  [:digit:]+ { return new Symbol(sym.NUMERO, new Integer(yytext())); }
32  [ \t\r\n]+              {;}
33  .           { System.out.println("Error léxico."+yytext()+"-"); }
```

Las acciones sobre la tabla de símbolos se han destacado en el listado del fichero **Calcuy.cup** que sería:

```
1   import java_cup.runtime.*;
2   import java.io.*;
3   parser code {:
4               static TablaSimbolos tabla = new TablaSimbolos();
5               public static void main(String[] arg){
6                   parser parserObj = new parser();
7                   Yylex miAnalizadorLexico =
8                               new Yylex(new InputStreamReader(System.in), tabla);
9                   parserObj.setScanner(miAnalizadorLexico);
10                  try{
11                      parserObj.parse();
12                      tabla.imprimir();
13                  }catch(Exception x){
14                      x.printStackTrace();
15                      System.out.println("Error fatal.\n");
16                  }
17              }
18  :};
19  terminal PUNTOYCOMA, MAS, MENOS, POR, ENTRE;
```

```
20 terminal UMENOS, LPAREN, RPAREN, PRINT, ASIG;
21 terminal Simbolo ID;
22 terminal Integer NUMERO;
23 non terminal listaExpr;
24 non terminal Integer expr, asig;
25 precedence left MAS, MENOS;
26 precedence left POR, ENTRE;
27 precedence left UMENOS;
28 listaExpr      ::=   listaExpr asig PUNTOYCOMA
29                           {: System.out.println("Asignacion(es) efectuada(s)."); :}
30                |     listaExpr PRINT expr:e PUNTOYCOMA
31                           {: System.out.println("= " + e); :}
32                |     listaExpr error PUNTOYCOMA
33                |     /* Epsilon */
34                ;
35 asig      ::=  ID:s ASIG expr:e  {:
36                                        RESULT = e;
37                                        s.valor = e;
38                                   :}
39                |     ID:s ASIG asig:e  {:
40                                        RESULT = e;
41                                        s.valor = e;
42                                   :}
43                ;
44 expr      ::=  expr:e1 MAS expr:e2
45                           {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
46           |    expr:e1 MENOS expr:e2
47                           {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
48           |    expr:e1 POR expr:e2
49                           {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
50           |    expr:e1 ENTRE expr:e2
51                           {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
52           |    ID:s             {: RESULT = s.valor; :}
53           |    NUMERO:n   {: RESULT = n; :}
54           |    MENOS expr:e     %prec UMENOS
55                           {: RESULT = new Integer(0 - e.intValue()); :}
56           |    LPAREN expr:e RPAREN   {: RESULT = e; :}
57           ;
```

6.4.4 Solución con JavaCC

El proceso que se sigue para solucionar el problema de la calculadora en JavaCC es casi igual al de JFlex/Cup solo que, recordemos, no es posible pasar más atributo del analizador léxico al sintáctico que el propio lexema leído de la entrada. De esta manera, hemos optado por insertar en la tabla de símbolos en la regla asociada al identificador, en lugar de en una acción léxica.

Lo más reseñable de esta propuesta es la regla asociada a la asignación múltiple:

```
asig    ::=     ( <ID> ":=" )+ expr() <PUNTOYCOMA>
```

Debido a la naturaleza descendente del analizador generado por JavaCC, resulta difícil propagar el valor de **expr** a medida que se construye el árbol, por lo que se ha optado por almacenar en un vector todos los objetos de tipo **Simbolo** a los que hay que asignar la expresión, y realizar esta operación de golpe una vez evaluada ésta. Una alternativa habría sido

```
asig    ::=     <ID> ":=" expr()
        |       <ID> ":=" asig()
```

que solucionaría el problema si **asig** devuelve el valor de la expresión. Nótese cómo esta solución pasa por convertir la gramática en recursiva a la derecha, aunque tiene el problema de ser LL(4) debido a que ambas reglas comienzan por <ID> ":=" y a que tanto **expr** como **asig** pueden comenzar con un <ID>.

Así pues, el programa en JavaCC **Calculadora.jj** es:

```
1  PARSER_BEGIN(Calculadora)
2      import java.util.*;
3      public class Calculadora{
4          static TablaSimbolos tabla = new TablaSimbolos();
5          public static void main(String args[]) throws ParseException {
6          new Calculadora(System.in).gramatica();
7              tabla.imprimir();
8          }
9      }
10 PARSER_END(Calculadora)
11 SKIP : {
12         " "
13     |   "\t"
14     |   "\n"
15     |   "\r"
16 }
17 TOKEN [IGNORE_CASE] :
18 {
19         <PRINT: "PRINT">
20     |   <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
21     |   <NUMERO: (["0"-"9"])+>
22     |   <PUNTOYCOMA: ";">
23 }
24 /*
25 gramatica ::= ( sentFinalizada )*
26 */
27 void gramatica():{}{
28     ( sentFinalizada() )*
29 }
30 /*
31 sentFinalizada ::=          PRINT expr ";"
32                         |   ( ID ":=" )+ expr ";"
33                         |   error ";"
34 */
```

```
35 void sentFinalizada():{
36       int resultado;
37       Vector v = new Vector();
38       Simbolo s;
39 }{
40       try{
41            <PRINT> resultado=expr() <PUNTOYCOMA>
42                                       { System.out.println("="+resultado); }
43       |
44            ( LOOKAHEAD(2)
45                 s=id() ":=" { v.add(s); } )+ resultado=expr() <PUNTOYCOMA>
46                                       {
47                                            Integer valor = new Integer(resultado);
48                                            Iterator it = v.iterator();
49                                            while(it.hasNext())
50                                                 ((Simbolo)it.next()).valor = valor;
51                                            System.out.println("Asignacion(es) efectuada(s).");
52                                       }
53       }catch(ParseException x){
54            System.out.println(x.toString());
55            Token t;
56            do {
57                 t = getNextToken();
58            } while (t.kind != PUNTOYCOMA);
59       }
60 }
61 /*
62 expr ::= term ( ( "+" | "-" ) term )*
63 */
64 int expr():{
65       int   acum1=0,
66             acum2=0;
67 }{
68       acum1=term()     (          ("+" acum2=term() {acum1+=acum2;} )
69                                |    ("-" acum2=term() {acum1-=acum2;} )
70                             )*
71       { return acum1; }
72 }
73 /*
74 term ::= fact ( ( "*" | "/" ) fact )*
75 */
76 int term():{
77       int   acum1=0,
78             acum2=0;
79 }{
80       acum1=fact()     (          ("*" acum2=fact() {acum1*=acum2;} )
81                                |    ("/" acum2=fact() {acum1/=acum2;} )
82                             )*
83       { return acum1; }
```

```
84 }
85 /*
86 fact ::= (“-“)* ( ID | NUMERO | "(" expr ")" )
87 */
88 int fact():{
89       int   acum=0,
90             signo=1;
91       Simbolo s;
92 }{
93       ("-" { signo *= -1; } )*
94       (           s=id() { acum = s.valor.intValue(); }
95             |     acum=numero()
96             |     "(" acum=expr() ")"
97       )
98       { return signo*acum; }
99 }
100 Simbolo id():{}{
101       <ID>      {
102                   Simbolo s;
103                   if ((s = tabla.buscar(token.image)) == null)
104                         s = tabla.insertar(token.image);
105                   return s;
106             }
107 }
108 int numero():{}{
109       <NUMERO>      { return Integer.parseInt(token.image); }
110 }
111 SKIP : {
112       <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
113 }
```

Como se ha comentado anteriormente, la necesidad de realizar todas las asignaciones de golpe en una asignación múltiple se podría evitar introduciendo la regla:

```
/*
asig ::= ( ID “:=” )+ expr
*/
int asig():{
            int resultado;
            Simbolo s;
}{
            (           LOOKAHEAD(4)
                        s=id() “:=” resultado=asig() { s.valor = new Integer(resultado); }
            |           s=id() “:=” resultado=expr() { s.valor = new Integer(resultado); }
            )
            { return resultado; }
}
```

Y la regla

```
id() ":=" )+ expr() <PUNTOYCOMA>
```

se sustituye por

```
asig() <PUNTOYCOMA> { System.out.println("Asignacion(es) efectuada(s)."); }
```

Capítulo 7

Gestión de tipos

7.1 Visión general

Un compilador debe comprobar si el programa fuente sigue tanto las convenciones sintácticas como las semánticas que describen al lenguaje fuente. Esta comprobación garantiza la detección y posterior comunicación al programador de ciertos errores de programación que facilitan la depuración pero que no pueden ser gestionados desde el punto de vista exclusivamente sintáctico. De este este tipo de errores, los más comunes se derivan de la utilización de tipos de datos, ya sean proporcionados por el propio lenguaje (tipos primitivos), o creados por el programador en base a construcciones sintácticas proporcionadas por el lenguaje (tipos de usuario). Básicamente, los tipos de datos sirven precisamente para que el programador declare con qué intención va a utilizar cada variable (para guardar un número entero, un número real, una cadena de caracteres, etc.), de manera que el compilador pueda detectar automáticamente en qué situaciones ha cometido un error el programador (por ejemplo, si ha intentado sumar un número entero y una cadena de caracteres).

Con la gestión de tipos se asegura que el tipo de una construcción sintáctica coincida con el previsto en su contexto. Por ejemplo, el operador aritmético predefinido **mod** en Modula-2 exige operandos numéricos enteros, de modo que mediante la gestión de tipos debe asegurarse de que dichos operandos sean de tipo entero. De igual manera, la gestión de tipos debe asegurarse de que la desreferenciación se aplique sólo a un puntero, de que la indización se haga sólo sobre una matriz, de que a una función definida por el usuario se aplique la cantidad y tipo de argumentos correctos, etc.

La información recogida por el gestor de tipos también es útil a la hora de generar código. Por ejemplo, los operadores aritméticos normalmente se aplican tanto a enteros como a reales, y tal vez a otros tipos, y se debe examinar el contexto de cada uno de ellos para determinar el espacio en memoria necesario para almacenar cálculos intermedios. Se dice que un símbolo o función está **sobrecargado** cuando representa diferentes operaciones en diferentes contextos; por ejemplo, la suma (expresada por el símbolo "+") suele estar sobrecargada para los números enteros y para los reales. La sobrecarga puede ir acompañada de una conversión de tipos, donde el compilador proporciona el operador para convertir un operando en el tipo esperado por el contexto. Considérense por ejemplo expresiones como "x + i" donde **x** es de tipo real e **i** es de tipo entero. Como la representación de enteros y reales es distinta dentro de un computador, y se utilizan instrucciones de máquina distintas para las operaciones sobre enteros y reales, puede que el compilador tenga que convertir primero uno de los operadores para garantizar que ambos operandos son del mismo tipo cuando tenga

lugar la suma. La gramática de la figura 7.1 genera expresiones formadas por constantes enteras y reales a las que se aplica el operador aritmético de la suma + . Cuando se suman dos enteros el resultado es entero, y si no es real. De esta manera, cada subexpresión tiene asociado un tipo y un espacio de almacenamiento dependiente de éste, al igual que tendrá asociado un valor en tiempo de ejecución, tal y como se ilustra en el árbol de la derecha de esta figura.

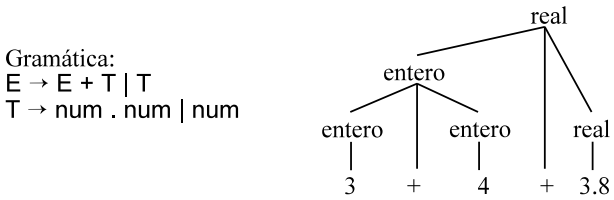


Figura 7.1. Ejemplo de cálculo del tipo de cada subexpresión. La suma de enteros produce un entero, pero si interviene un valor real, entonces el resultado será un real

Una noción diferente de la sobrecarga es la de **polimorfismo**, que aparece en los lenguajes orientados a objetos. En estos lenguajes existe una jerarquía de herencia en la que los tipos más especializados heredan de los tipos más generales, de manera que los tipos especializados pueden incorporar características de las que carecen los generales, e incluso pueden cambiar el funcionamiento de ciertos métodos de las clases ancestras en lo que se llama **reescritura de funciones**. En estos lenguajes, una función es polimórfica cuando tiene parámetros de tipo general, pero puede ejecutarse con argumentos de tipos especializados de la misma línea hereditaria, es decir, descendientes del tipo general, desencadenando en cada caso una acción diferente por medio de la **vinculación dinámica**. Según la vinculación dinámica, en tiempo de ejecución se detecta el tipo del parámetro real y se ejecutan sus métodos propios, en lugar de los del padre, si éstos están reescritos. Una cosa es que nuestro lenguaje tenga algunas funciones predefinidas sobrecargadas, y otra cosa que se permita la definición de funciones sobrecargadas. En este último caso, hay que solucionar de alguna forma los conflictos en la tabla de símbolos, ya que un mismo identificador de nombre de función tiene asociados diferentes parámetros y cuerpos.

7.2 Compatibilidad nominal, estructural y funcional

Cuando se va a construir un nuevo lenguaje de programación, resulta fundamental establecer un criterio riguroso y bien documentado de compatibilidad entre tipos, de tal manera que los programadores que vayan a utilizar ese lenguaje sepan a qué atenerse.

Entendemos que dos tipos **A** y **B** son compatibles cuando, dadas dos expresiones $\mathbf{e}_a$ y $\mathbf{e}_b$ de los tipos **A** y **B** respectivamente, pueden usarse indistintamente en un contexto determinado. Existen muchos contextos diferentes, de los cuales el más

usual es la asignación, de forma que si la variable $\mathbf{v_a}$ es de tipo **A** y se permite una asignación de la forma $\mathbf{v_a := e_b}$, diremos que **A** y **B** son compatibles para la asignación. Otro contexto viene dado por las operaciones aritméticas, como la suma: si un lenguaje admite expresiones de la forma $\mathbf{e_a + e_b}$, diremos que **A** y **B** son compatibles para la suma. En este último caso, los tipos **A** y **B** podrían representar, por ejemplo, los tipos **int** y **float** de C.

Básicamente se pueden establecer tres criterios de compatibilidad entre tipos: nominal, estructural y funcional. La compatibilidad nominal es la más estricta, esto es, la que impone las condiciones más fuerte para decidir cuándo dos variables se pueden asignar entre sí. La compatibilidad funcional es la más relajada, mientras que la estructural se halla en un nivel intermedio. Por supuesto, el desarrollador de compiladores puede establecer otros criterios que estén a medio camino entre cualesquiera de estas compatibilidades.

Para explicar la diferencia entre estos criterios usaremos los siguientes dos tipos definidos en C:

```
typedef struct{
        char nombre[20];
        char apellidos[35];
        int edad;
} empleado;
empleado e1, e2;
```

y

```
typedef struct{
        char nombre[20];
        char apellidos[35];
        int edad;
} cliente;
cliente c1, c2;
```

Los tipos **cliente** y **empleado** tienen diferente nombre, o sea, son diferentes desde el punto de vista nominal. Sin embargo, su contenido o estructura es la misma, luego son equivalentes desde el punto de vista estructural. De esta forma, un lenguaje con compatibilidad de tipos a nivel nominal impide hacer asignaciones como **e1=c1**, pero permite hacer **e1=e2**, ya que **e1** y **e2** pertenecen exactamente al mismo tipo (puesto que no puede haber dos tipos con el mismo nombre). Un lenguaje con compatibilidad de tipos estructural permite hacer ambas asignaciones, puesto que sólo requiere que la estructura del l-valor y del r-valor sea idéntica lo cual es cierto para los tipos **empleado** y **cliente**.

La compatibilidad nominal tiene fuertes implicaciones cuando se declaran variables de tipos anónimos. Siguiendo con el ejemplo, se tendría:

```
struct{
        char nombre[20];
        char apellidos[35];
```

```
        int edad;
} e1, e2;
```

y

```
struct{
        char nombre[20];
        char apellidos[35];
        int edad;
} c1, c2;
```

donde los tipos de las variables **e1**, **e2**, **c1** y **c2** carece de nombre. Por regla general, en estas situaciones, un lenguaje con compatibilidad de tipos nominal crea un tipo con un nombre interno para cada declaración anónima, lo que en el ejemplo se traduce en la creación de dos tipos, uno al que pertenecerán **e1** y **e2** (y que, por tanto, serán compatibles nominalmente), y otro al que pertenecerán **c1** y **c2** (que también serán compatibles nominalmente).

Por otro lado, la compatibilidad funcional ni siquiera requiere que las expresiones posean la misma estructura. Dos expresiones $\mathbf{e_a}$ y $\mathbf{e_b}$ de tipos **A** y **B** respectivamente son funcionalmente compatibles si **A** y **B** poseen diferente estructura pero pueden usarse indistintamente en algún contexto. El caso más usual consiste en poder sumar valores enteros y decimales. Por regla general, los enteros se suelen representar en complemento a dos, mientras que los números decimales se representan en coma flotante. Aunque se utilice el mismo número de octetos para representar a ambos, su estructura es esencialmente diferente, pero la mayoría de lenguajes de programación permite realizar operaciones aritméticas entre ellos. En estos lenguajes existe una compatibilidad funcional entre el tipo entero y el decimal.

La compatibilidad funcional requiere que el compilador detecte las situaciones en que se mezclan correctamente expresiones de tipos con diferente estructura, al objeto de generar código que realice la conversión del tipo de la expresión menos precisa a la de mayor precisión. Por ejemplo, al sumar enteros con decimales, los enteros se deben transformar a decimales y, sólo entonces, realizar la operación. A esta operación automática se la denomina **promoción** (del inglés *promotion*).

En los lenguajes orientados a objeto también aparece la compatibilidad por herencia: si una clase **B** hereda de otra **A** (**B** es un **A**), por regla general puede emplearse un objeto de clase **B** en cualquier situación en la que se espere un objeto de tipo **A**, ya que **B** es un **A** y por tanto cumple con todos los requisitos funcionales para suplantarlo.

7.3 Gestión de tipos primitivos

En este punto se estudiará la gestión de una calculadora básica, cuando en ésta se incluye el tipo cadena de caracteres, además del tipo entero con el que ya se ha trabajado en capítulos anteriores.

7.3.1 Gramática de partida

La gramática a reconocer permitirá gestionar valores de tipo entero y de tipo cadena de caracteres. Las cadenas se pueden concatenar mediante sobrecarga del operador "+" que, recordemos, también permite sumar enteros. Mediante dos funciones que forman parte del lenguaje se permitirá la conversión entre ambos tipos. Para evitar perdernos en detalles innecesarios, se han eliminado algunas operaciones aritméticas y la asignación múltiple. Además, para ilustrar que el retorno de carro no siempre ha de ser considerado como un separador, en la gramática que se propone se ha utilizado como terminador, haciendo las funciones del punto y coma de gramáticas de ejemplos anteriores. La gramática expresada en formato Yacc es la siguiente:

```
prog   :   prog asig '\n'
       |   prog IMPRIMIR expr '\n'
       |   prog error '\n'
       |   /* Épsilon */
       ;
asig   :   ID ASIG expr
       ;
expr   :   expr '+' expr
       |   expr '*' expr
       |   A_ENTERO '('expr ')'
       |   A_CADENA '(' expr ')'
       |   ID
       |   NUMERO
       |   CADENA
       ;
```

La misma gramática para JavaCC queda:

```
void gramatica():{}{
     ( sentFinalizada() )*
}
void sentFinalizada():{}{
          <IMPRIMIR> expr() <RETORNODECARRO>
     |    <ID><ASIG> expr() <RETORNODECARRO>
     |    /* Gestión de error */
}
void expr():{}{
     term() ( "+" term() )*
}
void term():{}{
     fact() ( "*" fact() )*
}
void fact():{}{
          <ID>
     |    <NUMERO>
     |    <CADENA>
     |    <A_CADENA> "(" expr() ")"
     |    <A_ENTERO> "(" expr() ")"
}
```

El cometido semántico de cada construcción es:

- Es posible asignar el valor de una expresión a una variable. No es posible hacer asignaciones múltiples: **ID ASIG expr**.
- Se puede trabajar con valores de dos tipos: cadenas de caracteres y números enteros.
- La gramática no posee una zona donde el programador pueda declarar el tipo de cada variable, sino que éste vendrá definido por el de la primera asignación válida que se le haga. Es decir, si la primera vez que aparece la variable x se le asigna el valor 65, entonces el tipo de x es entero. El tipo de una variable no puede cambiar durante la ejecución de un programa.
- Es posible visualizar el valor de una expresión: **IMPRIMIR expr**. Esta sentencia permite, por tanto, visualizar tanto valores enteros como cadenas de caracteres, por lo que si la consideramos desde el punto de vista funcional, podemos decir que **IMPRIMIR** es una función polimórfica.
- Con los enteros vamos a poder hacer sumas y multiplicaciones. En cambio con las cadenas no tiene demasiado sentido hacer ninguna de estas operaciones, aunque utilizaremos el símbolo de la suma para permitir concatenaciones de cadenas. No es posible sumar o concatenar enteros y cadenas.
- Es posible realizar conversiones de tipos mediante funciones propias del lenguaje de programación. Éstas son diferentes a las funciones preconstruidas (built-ins), que se encuentran almacenadas en librerías pero cuyos nombres no forman realmente parte de la sintaxis del lenguaje. Estas funciones son:
  - **A_CADENA(expr)**: tiene como parámetro una expresión de tipo entero y nos devuelve como resultado la cadena de texto que la representa.
  - **A_ENTERO(expr)**: tiene como parámetro una expresión de tipo carácter que representa un número y nos devuelve como resultado su valor entero.

7.3.2 Pasos de construcción

Una vez que se tiene clara la gramática que se va a usar, es conveniente seguir los pasos que se indican a continuación a fin de culminar la construcción del traductor. En estos pasos suele ser común tener que volver atrás con el fin de reorganizar la gramática para adaptar sus construcciones a los requerimientos de los metacompiladores.

7.3.2.1 Propuesta de un ejemplo

Con objeto de proseguir los pasos de construcción, suele ser buena idea plantear un ejemplo de entrada al traductor y la salida que se desea obtener. La entrada aparece en negro, la salida en azul y los mensajes de error en rojo:

❶ b := “Camarada Trotski”
❷ a :=7
❸ c :=12 + 3

❹ PRINT c + 5
20
❺ PRINT b
Camarada Trotski
❻ PRINT b+7
No se pueden sumar cadenas y enteros.
❼ PRINT b+", fundador de la URSS."
Camarada Trotski, fundador de la URSS.
❽ PRINT A_ENTERO("23")*4
92

7.3.2.2 Definición de la tabla de símbolos

En este paso se define la estructura de la tabla de símbolos, si es que esta es necesaria para el problema, así como su interfaz.

En este caso sí necesitaremos la tabla de símbolos, ya que en ella se almacenarán los valores de los identificadores. También será necesario guardar el tipo de cada variable con objeto de controlar que su utilización sea coherente con las intenciones de su declaración (implícita en la primera asignación). De esta forma, el tipo de una variable se guardará en la tabla de símbolos mediante un carácter con el siguiente significado:

- 'c': tipo cadena de caracteres.
- 'e': tipo entero.
- 'i': tipo indefinido. Este tipo se usará en caso de que en la primera asignación se le intente dar a una variable el valor resultante de una expresión errónea.

Parece claro que, dado que existen variables de tipos diferentes, una entrada de la tabla de símbolos debe ser capaz de guardar tanto una cadena como un entero, en función del tipo de la variable que representa. En lenguaje C esto se soluciona mediante una estructura **union**, mientras que en Java se tendrá un manejador a **Object** que, en función del tipo de la variable, apuntará a un **Integer** o a un **String**. La figura 7.2 muestra gráficamente la estructura de la tabla de símbolos.

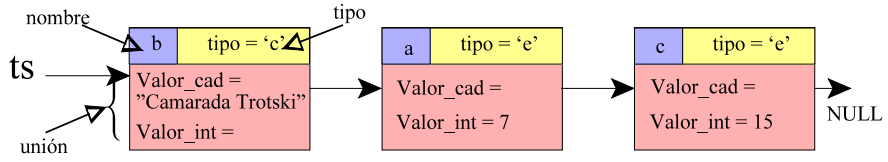


Figura 7.2. Situación de la tabla de símbolos tras introducir en la calculadora las sentencias del punto 7.3.2.1

7.3.2.3 Asignación de atributos

El siguiente paso consiste en estudiar qué atributo se le tiene que asociar a los terminales y no terminales.

7.3.2.3.1 Atributos de terminales

Con respecto a los terminales **NUMERO** y **CADENA**, dado que representan constantes enteras y de cadena de caracteres, lo más sensato es asignarles como atributos las constantes a que representan. En el caso de **NUMERO** se realizará una conversión del lexema leído al número entero correspondiente, dado que nos interesa operar aritméticamente con él. En cuanto a la cadena de caracteres, habrá que eliminar las comillas de apertura y de cierre ya que son sólo delimitadores que no forman parte del literal en sí.

Por otro lado, y como ya se ha comentado, dado que estamos definiendo un lenguaje sin zona de declaraciones explícitas, será el analizador léxico quien inserte los identificadores en la tabla de símbolos, con objeto de descargar al sintáctico de algunas tareas. El atributo del terminal **ID** es, por tanto, una entrada de dicha tabla. Básicamente, el analizador lexicográfico, cuando se encuentra con un identificador, lo busca en la tabla; si está asocia su entrada como atributo al **ID** encontrado y retorna al sintáctico; si no está crea una nueva entrada en la tabla y la retorna al sintáctico. El problema que se plantea aquí es: ¿con qué se rellena una nueva entrada en la tabla de símbolos? Analicemos por ejemplo la primera sentencia del apartado 7.3.2.1:

b = “Camarada Trotski”

Cuando el analizador léxico encuentra el identificador **b**, lo busca en la tabla de símbolos (que inicialmente está vacía) y, al no encontrarlo, crea un nuevo nodo en ésta. Este nodo tendrá como nombre a “b”, pero ¿cuál es su tipo? El tipo de **b** depende del valor que se le dé en la primera asignación (en este caso una cadena de caracteres), pero el problema es que el analizador léxico no ve más allá del lexema en el que se encuentra en este momento, “b”, por lo que aún es incapaz de conocer el tipo de la expresión que hay detrás del terminal de asignación. El tipo de **b** está implícito en el momento en que se hace su primera asignación, pero el momento en que el analizador léxico realiza la inserción en la tabla de símbolos está antes del momento en que se conoce el tipo de la expresión asignada. La solución es que el analizador sintáctico será el que le ponga el tipo cuando se concluya la primera asignación; mientras tanto, el lexicográfico indicará un tipo transitorio indefinido: “i”. La figura 7.3 muestra el contenido de la tabla de símbolos en el preciso instante en que el analizador léxico retorna al sintáctico la entrada de la tabla de símbolos correspondiente a la **c** de la sentencia ❸.

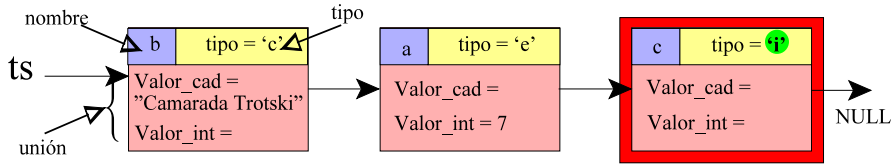


Figura 7.3. La entrada de la tabla marcada en rojo acaba de ser insertada por el analizador léxico que, como desconoce el tipo de la variable recién insertada **c**, la pone como indefinida: **i**

7.3.2.3.2 Atributos de no terminales

Hasta ahora, en nuestro ejemplo de la calculadora, el no terminal **expr** tenía como atributo el valor entero a que representaba. Sin embargo, en el caso que ahora nos ocupa, el concepto de expresión denotado por el no terminal **expr** sirve para representar expresiones tanto enteras como de cadena de caracteres. Esta dualidad queda bien implementada mediante el concepto de **union** en C, de manera parecida a como se ha hecho en los nodos de la tabla de símbolos. De esta manera, el atributo de **expr** es un registro con un campo para indicar su tipo y una **union** para indicar su valor, ya sea entero o de cadena. En el caso de Java, la solución es aún más sencilla ya que el atributo puede ser un simple **Object**; si dicho **Object** es **instanceof Integer** el tipo será entero; si el **Object** es **instanceof String** el tipo será cadena de caracteres; y si el **null** el tipo será indefinido. En cualquier caso, en aras de una mayor legibilidad, también es posible asociar un objeto con dos campos, tipo y valor, como en la solución en C. La figura 7.5 muestra un ejemplo de los atributos asociados cuando se reconoce la sentencia ❸ del apartado 7.3.2.1. En este caso en concreto, el atributo de **expr** debe coincidir conceptualmente con parte de la estructura de un nodo de la tabla de símbolos, tal y como se muestra en la figura 7.4.

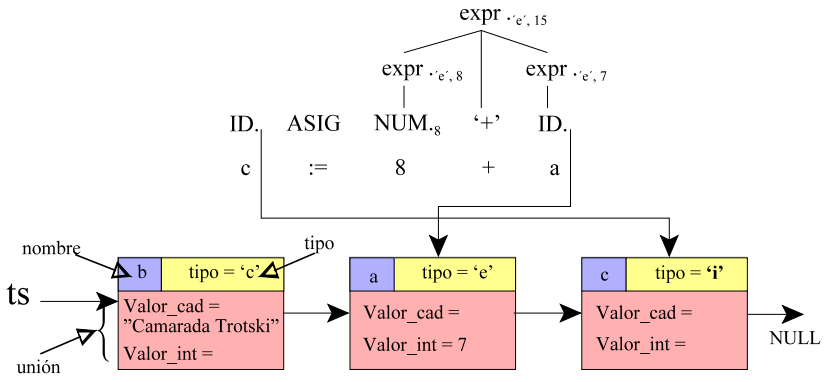


Figura 7.5. Reconocimiento de una sentencia de asignación y atributos asociados a los terminales y no terminales del árbol sintáctico

El atributo asociado a una expresión permite manipular los errores semánticos. Por ejemplo, una sentencia de la forma:

c = 8 + "hola"

es sintácticamente correcta ya que existe un árbol sintáctico que la reconoce, pero es

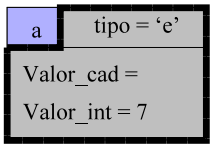


Figura 7.4. El atributo de una expresión es un tipo y un valor (entero o de cadena en función del tipo), lo que coincide con parte de un nodo de la tabla de símbolos (zona sombreada)

incorrecta semánticamente, ya que no vamos a permitir mezclar números y cadenas en una suma. Por tanto, la expresión resultante en el árbol sintáctico tendrá como tipo el valor 'i' de indefinido, tal y como se muestra en el árbol de la figura 7.6.

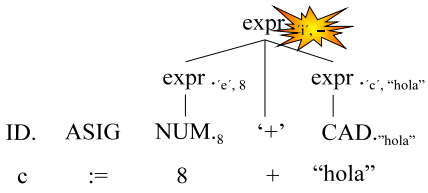


Figura 7.6. Se produce un error semántico porque no se puede sumar un entero y una cadena. La expresión resultante es, por tanto, indefinida

7.3.2.4 Acciones semánticas

En este apartado se estudiarán las acciones semánticas desde dos puntos de vista:

- Los controles de compatibilidad de tipos.
- Las acciones a realizar en caso de que se satisfagan las restricciones de tipos.

Por regla general, el estudio de las acciones semánticas suele hacerse comenzando por las reglas por las que primero se reduce en un árbol sintáctico cualquiera, ya que éstas suelen ser las más sencillas, y así se aborda el problema en el mismo orden en el que se acabarán ejecutando las acciones semánticas. Las reglas:

```
expr    :   NUMERO
        |   CADENA
```

son las más simples ya que la acción semántica asociada tan sólo debe asignar el tipo a **expr** y copiar el atributo del *token* como valor de **expr**.

La regla:

```
expr    :   expr '*' expr
```

debe controlar que las dos expresiones sean de tipo entero. Si lo son, el tipo de la **expr** resultante también es entero y el valor será el producto de las expresiones del consecuente. En caso contrario, resultará una expresión de tipo indefinido.

La regla:

```
expr    :   expr '+' expr
```

es un poco más complicada, ya que el operador '+' está sobrecargado y con él puede tanto sumarse enteros como concatenarse cadenas de caracteres. Por tanto, la acción semántica debe comprobar que los tipos de las expresiones del antecedente sean iguales. Si son enteras se produce una expresión entera suma de las otras dos. Si son cadenas de caracteres se produce una expresión que es la concatenación de las otras dos. En cualquier otro caso se produce una expresión indefinida.

Las reglas:

```
expr    :   A_ENTERO '('expr ')'
```

```
        |   A_CADENA '(' expr ')'
```

toman, respectivamente, expresiones de tipo cadena y entero y devuelven enteros y cadenas, también respectivamente. En cualquier otro caso se produce una expresión indefinida. La función **A_CADENA()** convierte el valor de su parámetro en una ristra de caracteres que son dígitos. La función **A_ENTERO()** convierte su parámetro en un número, siempre y cuándo la cadena consista en una secuencia de dígitos; en caso contrario también se genera una expresión indefinida.

La regla:

```
expr    :   ID
```

es trivial, ya que el analizador sintáctico recibe del léxico la entrada de la tabla de símbolos en la que reside la variable. Por tanto, dado que el atributo de una expresión comparte parte de la estructura de un nodo de la tabla de símbolos, en esta acción semántica lo único que hay que hacer es copiar dicho trozo (ver figura 7.4).

La semántica de la asignación dada por la regla:

```
asig    :   ID ASIG expr
```

es un poco más compleja. Recordemos que el tipo de una variable viene dado por el tipo de lo que se le asigna por primera vez. Por ejemplo, si en laprimera asignación a la variable **a** se le asigna el valor 7, entonces su tipo es entero durante toda la ejecución del programa; si se le asigna "Ana", entonces será de tipo cadena. Por tanto, la asignación debe:

1. Si el tipo del l-valor es indefinido, se le asigna el valor y el tipo del r-valor, sea éste el que sea.
2. Si el tipo del l-valor está definido, entonces:
   2.1. Si coinciden los tipos del l-valor y del r-valor entonces se asigna al l-valor el valor del r-valor.
   2.2. Si no coinciden los tipos, se emite un mensaje de error y el l-valor se deja intacto.

En esta explicación queda implícito qué hacer cuando se intenta asignar a una variable una expresión indefinida: si la variable aún es indefinida se deja tal cual; si no, la variable no pasa a valer indefinido, sino que se queda como está. Si ante una asignación indefinida se nos ocurriera poner la variable en estado indefinido, la secuencia:

```
a = 7              ✔ Funciona
a = 7 + "hola"     ✗ Falla (y cambia el tipo de a a indefinido)
a = "mal"          ✔ Funciona (pero no debería)
```

cambiaría el valor y el tipo de **a**, lo cual es incorrecto con respecto a la semántica de nuestro lenguaje.

Una vez asignado el tipo a una variable éste siempre es el mismo; si durante la ejecución se le asigna una expresión errónea, el tipo de la variable seguirá siendo el mismo y su valor tampoco cambiará. Este comportamiento podría modificarse añadiendo una variable *booleana* a la tabla de símbolos que me diga si el valor de la variable es conocido o no, de tal manera que una asignación errónea pondría esta

variable a *false*, tal y como se ilustra en la figura 7.7.

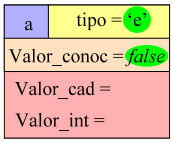


Figura 7.7. Inclusión de un campo que indica que, a pesar de conocerse el tipo de la variable, su valor se desconoce debido a que la última asignación fue errónea

7.3.3 Solución con Lex/Yacc

La solución en C parte de una tabla de símbolos con la estructura propuesta en los apartados anteriores, y con la misma interfaz que la que se introdujo en el apartado 6.4.2 y que se muestra a continuación en el fichero **TabSimb.c**:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef struct _simbolo {
4        struct _simbolo * sig;
5        char nombre[20];
6        char tipo;
7        union {
8              char valor_cad[100];
9              int valor_int;
10       } info;
11 } simbolo;
12
13 void insertar(simbolo ** p_t, simbolo * s) {
14       s->sig = (*p_t);
15       (*p_t) = s;
16 }
17
18 simbolo * buscar(simbolo * t, char nombre[20]) {
19       while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
20             t = t->sig;
21       return (t);
22 }
23 void imprimir(simbolo * t) {
24       while (t != NULL) {
25             printf("%s:%c: ", t->nombre, t->tipo);
26             if(t->tipo == 'c') printf("%s\n", t->info.valor_cad);
27             else if(t->tipo == 'e') printf("%d\n", t->info.valor_int);
28             else printf("Indefinido\n");
29             t = t->sig;
30       }
31 }
```

El programa Lex funciona de manera parecida al de la calculadora del capítulo 6, sólo que ahora hemos de reconocer también literales de tipo cadena de caracteres que se retornan a través del *token* **CADENA**. Como atributo se le asocia el lexema sin

comillas. Además, cuando se reconoce un identificador de usuario por primera vez, se debe introducir en la tabla de símbolos indicando que su tipo es indefinido, de manera que el sintáctico pueda asignarle el tipo correcto cuando éste se conozca a través del tipo y valor de la primera expresión que se le asigne. Además, el retorno de carro no se ignora sino que se le devuelve al analizador sintáctico ya que forma parte de nuestra gramática. Por último, nótese que el *token* **IMPRIMIR** se corresponde con el lexema "PRINT", puesto que nada obliga a que los *tokens* deban llamarse igual que las palabras reservadas que representan. El programa **TipSimpl.lex** queda:

```
1  %%
2  [0-9]+    {
3                  yylval.numero=atoi(yytext);
4                  return NUMERO;
5            }
6  \"[^\"]*\"  {
7                  strcpy(yylval.cadena, yytext+1);
8                  yylval.cadena[yyleng-2] = 0;
9                  return CADENA;
10           }
11 "PRINT"            { return IMPRIMIR;}
12 "A_ENTERO"         { return A_ENTERO;}
13 "A_CADENA"         { return A_CADENA;}
14 ":="               { return ASIG; }
15 [a-zA-Z][a-zA-Z0-9]*   {
16                              if (strlen(yytext)>19) yytext[19] = 0;
17                              yylval.ptr_simbolo = buscar(t,yytext);
18                              if(yylval.ptr_simbolo == NULL) {
19                                   yylval.ptr_simbolo = (simbolo *)malloc(sizeof(simbolo));
20                                   strcpy(yylval.ptr_simbolo->nombre,yytext);
21                                   yylval.ptr_simbolo->tipo='i';
22                                   insertar(&t,yylval.ptr_simbolo);
23                              }
24                              return ID;
25                         }
26 [ \t]+      {;}
27 .|\n        { return yytext[0]; }
```

La línea 16 ha sido incluida para evitar errores graves en C cuando el programador utilice nombres de variables de más de 19 caracteres.

Por último, el programa Yacc realiza exactamente las acciones comentadas en el apartado 7.3.2.4. El control que exige que la cadena que se convierta en entero deba estar formada sólo por dígitos (con un guión opcional al principio para representar los negativos) se realiza mediante la función **esNumero()** definida en la sección de código C del área de definiciones. Además, se ha prestado especial cuidado en que los errores semánticos de tipos no produzcan más que un sólo mensaje por pantalla. Todos nos hemos enfrentado alguna vez a un compilador que, ante un simple error en una línea de código, proporciona tres o cuatro mensajes de error. Estas cadenas de errores dejan perplejo al programador, de manera que el constructor de compiladores debe

evitarlas en la medida de sus posibilidades. De esta forma, ante una entrada como "c := 4 * "p" * "k" * 5", cuyo árbol se presenta en la figura 7.8.a) sólo se muestra un mensaje de error. Ello se debe a que los mensajes de error al sumar o multiplicar se muestran sólo cuando cada parámetro es correcto por sí sólo, pero operado con el adyacente produce un error. No obstante, una entrada como "c := 4 + "p" + "k" * 5", cuyo árbol se presenta en la figura 7.8.b), posee dos errores de tipos: 1) la suma de 4 más "p"; y 2) el producto de "k" por 5; nuestro intérprete informa de los dos, ya que se producen en ramas diferentes del árbol.

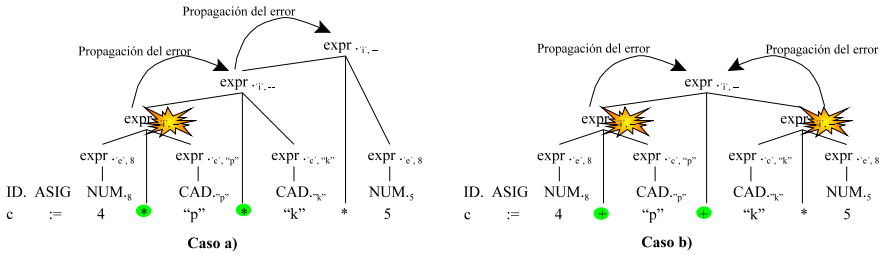


Figura 7.8. Gestión de errores de tipos

El fichero **TipSimpy.yac** es:

```
%{
#include "TabSimb.c"
typedef struct {
        char tipo;
        union {
                char valor_cad[100];
                int valor_int;
        } info;
} expresion;
int esNumero(char * s){
        int i=0;
        if (s[i] == '-') i++;
        for(; s[i] != 0; i++)
             if ((s[i]<'0') || (s[i]>'9')) return 0;
        return 1;
}
simbolo * t = NULL;
%}

%union{
        char cadena[100];
        int numero;
        simbolo * ptr_simbolo;
        expresion valor;
}
%token <cadena> CADENA
```

```
27 %token <numero> NUMERO
28 %token <ptr_simbolo> ID
29 %token IMPRIMIR ASIG A_ENTERO A_CADENA
30 %type <valor> expr
31 %start prog
32 %left '+'
33 %left '*'
34 %%
35 prog    :    /* Épsilon */
36          |    prog asig '\n'
37          |    prog IMPRIMIR expr '\n'{
38                       if ($3.tipo == 'e')
39                         printf("%d\n",$3.info.valor_int);
40                       else if ($3.tipo == 'c')
41                         printf("%s\n",$3.info.valor_cad);
42                       else
43                         printf("Indefinido.\n");
44                   }
45          |    prog error '\n'      { yyerrok; }
46          ;
47 asig    :    ID ASIG expr {
48                       if($1->tipo == 'i')
49                           $1->tipo = $3.tipo;
50                       if ($3.tipo == 'i')
51                           printf("Asignacion no efectuada.\n");
52                       else
53                           if ($3.tipo != $1->tipo)
54                               printf("Asignacion de tipos incompatibles.\n");
55                           else if($1->tipo =='e')
56                                   $1->info.valor_int = $3.info.valor_int;
57                           else
58                                   strcpy($1->info.valor_cad, $3.info.valor_cad);
59                   }
60      ;
61 expr    :    expr '+' expr {
62                       if(($1.tipo == 'c') && ($3.tipo == 'c')) {
63                           $$.tipo = 'c';
64                           sprintf($$.info.valor_cad,
65                                   "%s%s", $1.info.valor_cad, $3.info.valor_cad);
66                       } else if(($1.tipo == 'e') && ($3.tipo == 'e')) {
67                           $$.tipo = 'e';
68                           $$.info.valor_int = $1.info.valor_int + $3.info.valor_int;
69                       } else {
70                           $$.tipo = 'i';
71                           if (($1.tipo != 'i') && ($3.tipo != 'i'))
72                               printf("No se pueden sumar cadenas y enteros.\n");
73                       }
74                   }
75          |    expr '*' expr  {
```

```
76                          if(($1.tipo == 'c') || ($3.tipo == 'c')) {
77                              $$.tipo = 'i';
78                              printf("Una cadena no se puede multiplicar.\n");
79                          } else if(($1.tipo == 'e') || ($3.tipo == 'e')) {
80                              $$.tipo = 'e';
81                              $$.info.valor_int = $1.info.valor_int * $3.info.valor_int;
82                          } else
83                              $$.tipo = 'i';
84                      }
85          |       A_ENTERO '(' expr ')'{
86                          if ($3.tipo != 'c') {
87                              $$.tipo = 'i';
88                              printf("Error de conversión. Se requiere una cadena.\n");
89                          } else if (esNumero($3.info.valor_cad)){
90                              $$.tipo = 'e';
91                              $$.info.valor_int= atoi($3.info.valor_cad);
92                          } else{
93                              $$.tipo = 'i';
94                              printf("La cadena a convertir sólo puede tener dígitos.\n");
95                          }
96                      }
97          |       A_CADENA '(' expr ')'       {
98                          if ($3.tipo != 'e') {
99                              $$.tipo = 'i';
100                             printf("Error de conversión. Se requiere un entero.\n");
101                         } else {
102                             $$.tipo = 'c';
103                             itoa($3.info.valor_int, $$.info.valor_cad, 10);
104                         };
105                     }
106         |       ID  {
107                         $$.tipo = $1->tipo;
108                         if ($$.tipo == 'i')
109                             printf("Tipo de %s no definido.\n",$1->nombre);
110                         else
111                             if ($$.tipo == 'e')
112                                 $$.info.valor_int = $1->info.valor_int;
113                             else
114                                 strcpy($$.info.valor_cad, $1->info.valor_cad);
115                     }
116         |       NUMERO    {
117                         $$.tipo = 'e';
118                         $$.info.valor_int = $1;
119                     }
120     | CADENA    {
121                         $$.tipo = 'c';
122                         strcpy($$.info.valor_cad, $1);
123                     }
124         ;
```

```
125 %%
126 #include "TipSimpl.c"
127 void main(){
128       yyparse();
129       imprimir(t);
130 }
131 void yyerror(char * s){
132       printf("%s\n",s);
133 }
```

7.3.4 Solución con JFlex/Cup

La solución con estas dos herramientas obedece a los mismos mecanismos que la anterior, con Lex/Yacc. Quizás la diferencia más notable sea el tratamiento del tipo de una expresión, o de una variable. En Java aprovechamos la clase **Object** para decir que el atributo de una expresión será un **Object** que apuntará a un **Integer**, a un **String** o a **null** en caso de que el valor sea indefinido. La función **tipo()** declarada en el ámbito de las acciones semánticas de Cup produce el carácter 'e', 'c' o 'i' en función de aquéllo a lo que apunte realmente un atributo.

Por otro lado, en la función **A_ENTERO()** no necesitamos una función en Java que nos diga si la cadena que se le pasa como parámetro tiene formato de número o no, ya que ello puede conocerse mediante la captura de la excepción **NumberFormatException** en el momento de realizar la conversión.

Así, el fichero **TablaSimbolos.java** es:

```
1  import java.util.*;
2  class Simbolo{
3        String nombre;
4        Object valor;
5        public Simbolo(String nombre, Object valor){
6            this.nombre = nombre;
7            this.valor = valor;
8        }
9  }
10 public class TablaSimbolos{
11       HashMap t;
12       public TablaSimbolos(){
13           t = new HashMap();
14       }
15       public Simbolo insertar(String nombre){
16           Simbolo s = new Simbolo(nombre, null);
17           t.put(nombre, s);
18           return s;
19       }
20       public Simbolo buscar(String nombre){
21           return (Simbolo)(t.get(nombre));
22       }
23       public void imprimir(){
```

```
24              Iterator it = t.values().iterator();
25              while(it.hasNext()){
26                   Simbolo s = (Simbolo)it.next();
27                   System.out.print(s.nombre + ": ");
28                   if (s.valor == null) System.out.print("i: Indefinido");
29                   else if (s.valor instanceof Integer) System.out.print("e: "+s.valor.toString());
30                   else if (s.valor instanceof String) System.out.print("c: "+s.valor);
31              }
32         }
33   }
```

El fichero **TipSimb.jflex** queda:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  %%
4  %{
5        private TablaSimbolos tabla;
6        public Yylex(Reader in, TablaSimbolos t){
7             this(in);
8             this.tabla = t;
9        }
10 %}
11 %unicode
12 %cup
13 %line
14 %column
15 %%
16 "+"        { return new Symbol(sym.MAS); }
17 "*"        { return new Symbol(sym.POR); }
18 "("        { return new Symbol(sym.LPAREN); }
19 ")"        { return new Symbol(sym.RPAREN); }
20 "\n"       { return new Symbol(sym.RETORNODECARRO); }
21 ":="       { return new Symbol(sym.ASIG); }
22 \"[^\"]*\"  { return new Symbol(sym.CADENA,yytext().substring(1,yytext().length()-1));}
23 [:digit:]+ { return new Symbol(sym.NUMERO, new Integer(yytext())); }
24 "PRINT"                  { return new Symbol(sym.IMPRIMIR); }
25 "A_CADENA"              { return new Symbol(sym.A_CADENA); }
26 "A_ENTERO"              { return new Symbol(sym.A_ENTERO); }
27 [:jletter:][:jletterdigit:]*  {
28                                   Simbolo s;
29                                   if ((s = tabla.buscar(yytext())) == null)
30                                        s = tabla.insertar(yytext());
31                                   return new Symbol(sym.ID, s);
32                              }
33 [ \t\r]+   {;}
34 .          { System.out.println("Error léxico."+yytext()+"-"); }
```

Y por último, **TipSimp.cup** es:

```
1  import java_cup.runtime.*;
2  import java.io.*;
```

```
3  parser code {:
4        static TablaSimbolos tabla = new TablaSimbolos();
5        public static void main(String[] arg){
6              parser parserObj = new parser();
7              Yylex miAnalizadorLexico =
8                           new Yylex(new InputStreamReader(System.in), tabla);
9              parserObj.setScanner(miAnalizadorLexico);
10             try{
11                   parserObj.parse();
12                   tabla.imprimir();
13             }catch(Exception x){
14                   x.printStackTrace();
15                   System.out.println("Error fatal.\n");
16             }
17       }
18 :};
19 action code {:
20       private static char tipo(Object o){
21             if (o == null) return 'i';
22             else if (o instanceof Integer) return 'e';
23             else return 'c';
24       }
25 :}
26 terminal RETORNODECARRO, MAS, POR;
27 terminal IMPRIMIR, ASIG, LPAREN, RPAREN, A_ENTERO, A_CADENA;
28 terminal Simbolo ID;
29 terminal Integer NUMERO;
30 terminal String CADENA;
31 non terminal asig, prog;
32 non terminal Object expr;
33 precedence left MAS;
34 precedence left POR;
35 /* Gramática */
36 prog      ::=   /* Épsilon */
37                 |     prog asig RETORNODECARRO
38                 |     prog IMPRIMIR expr:e RETORNODECARRO {:
39                             if (tipo(e) == 'i')
40                               System.out.println("Indefinido.");
41                             else
42                               System.out.println(e.toString());
43                       :}
44                 | prog error RETORNODECARRO
45                 ;
46 asig      ::= ID:s ASIG expr:e {:
47                       if (tipo(e) == 'i')
48                             System.out.println("Asignacion no efectuada.");
49                       else
50                             if ((s.valor == null) || (tipo(s.valor) == tipo(e)))
51                                   s.valor = e;
```

```
52                          else
53                              System.err.println("Asignacion de tipos incompatibles.");
54                  :}
55          ;
56  expr    ::=   expr:e1 MAS expr:e2 {:
57                  if((tipo(e1) == 'c') && (tipo(e2) == 'c'))
58                      RESULT = e1.toString() + e2.toString();
59                  else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
60                      RESULT = new Integer(     ((Integer)e1).intValue()
61                                                + ((Integer)e2).intValue());
62                  else {
63                      RESULT = null;
64                      if ((tipo(e1) != 'i') && (tipo(e2) != 'i'))
65                          System.err.println("No se pueden sumar cadenas y enteros.");
66                  }
67          :}
68          |     expr:e1 POR expr:e2 {:
69                          if((tipo(e1) == 'c') || (tipo(e2) == 'c')) {
70                              RESULT = null;
71                              System.err.println("Una cadena no se puede multiplicar.");
72                          } else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
73                              RESULT = new Integer(     ((Integer)e1).intValue()
74                                                        * ((Integer)e2).intValue());
75                          else
76                              RESULT = null;
77                      :}
78          | A_ENTERO LPAREN expr:e RPAREN {:
79                  if (tipo(e) != 'c') {
80                      RESULT = null;
81                      System.err.println("Error de conversión. Se requiere una cadena.");
82                  } else try {
83                      RESULT = new Integer(Integer.parseInt(e.toString()));
84                  } catch (Exception x){
85                      RESULT = null;
86                      System.err.println("La cadena a convertir sólo puede tener dígitos.");
87                  }
88          :}
89          | A_CADENA LPAREN expr:e RPAREN {:
90                          if (tipo(e) != 'e') {
91                              RESULT = null;
92                              System.err.println("Error de conversión. Se requiere un entero.");
93                          } else
94                              RESULT = e.toString();
95                      :}
96          | ID:s      {:
97                          RESULT = s.valor;
98                          if (tipo(s.valor) == 'i')
99                              System.err.println("Tipo de "+ s.nombre +" no definido.");
100                 :}
```

```
101         | NUMERO:n      {: RESULT = n; :}
102         | CADENA:c      {: RESULT = c; :}
103         ;
```

7.3.5 Solución con JavaCC

Construir la solución en JavaCC una vez solucionado el problema con JFlex/Cup resulta muy sencillo. Básicamente se han seguido los siguientes pasos:

- Crear la clase contenedora con la función **main()**.
- Reconocer los *tokens* necesarios.
- Incluir la gramática del punto 7.3.1.
- Incluir variables en las reglas para almacenar los atributos retornados por los no terminales.
- Las acciones semánticas a incluir son las mismas que en JFlex/Cup; en este caso hemos optado por incluir estas acciones en funciones propias de la clase **Calculadora**, sólo que cambiando cada asignación a **RESULT** de la forma "RESULT = dato;" por "return dato;" y haciendo que ésta sea la última sentencia de cada acción. El extraer las acciones en funciones aparte aumenta la legibilidad y claridad del código, aunque se pierde la localidad espacial de las reglas y sus acciones.

El programa **Calculadora.jj** es:

```
1  PARSER_BEGIN(Calculadora)
2      import java.util.*;
3      public class Calculadora{
4          static TablaSimbolos tabla = new TablaSimbolos();
5          public static void main(String args[]) throws ParseException {
6          new Calculadora(System.in).gramatica();
7              tabla.imprimir();
8          }
9          private static char tipo(Object o){
10             if (o == null) return 'i';
11             else if (o instanceof Integer) return 'e';
12             else return 'c';
13         }
14         private static void usarIMPRIMIR(Object e){
15             if (tipo(e) == 'i')
16               System.out.println("Indefinido.");
17             else
18               System.out.println(e.toString());
19         }
20         private static void usarASIG(Simbolo s, Object e){
21             if (tipo(e) == 'i')
22                 System.out.println("Asignacion no efectuada.");
23             else
24                 if ((s.valor == null) || (tipo(s.valor) == tipo(e)))
```

```
25                          s.valor = e;
26                      else
27                          System.err.println("Asignacion de tipos incompatibles.");
28          }
29          private static Object usarMAS(Object e1, Object e2){
30              if((tipo(e1) == 'c') && (tipo(e2) == 'c'))
31                  return e1.toString() + e2.toString();
32              else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
33                  return new Integer(((Integer)e1).intValue() + ((Integer)e2).intValue());
34              else {
35                  if ((tipo(e1) != 'i') && (tipo(e2) != 'i'))
36                      System.err.println("No se pueden sumar cadenas y enteros.");
37                  return null;
38              }
39          }
40          private static Object usarPOR(Object e1, Object e2){
41              if((tipo(e1) == 'c') || (tipo(e2) == 'c')) {
42                  System.err.println("Una cadena no se puede multiplicar.");
43                  return null;
44              } else if((tipo(e1) == 'e') && (tipo(e2) == 'e'))
45                  return new Integer(((Integer)e1).intValue() * ((Integer)e2).intValue());
46              else
47                  return null;
48          }
49          private static Object usarID(Simbolo s){
50              if (tipo(s.valor) == 'i')
51                  System.err.println("Tipo de "+ s.nombre +" no definido.");
52              return s.valor;
53          }
54          private static Object usarA_CADENA(Object e){
55              if (tipo(e) != 'e') {
56                  System.err.println("Error de conversión. Se requiere un entero.");
57                  return null;
58              } else
59                  return e.toString();
60          }
61          private static Object usarA_ENTERO(Object e){
62              if (tipo(e) != 'c') {
63                  System.err.println("Error de conversión. Se requiere una cadena.");
64                  return null;
65              } else try {
66                  return new Integer(Integer.parseInt(e.toString()));
67              } catch (Exception x){
68                  System.err.println("La cadena a convertir sólo puede tener dígitos.");
69                  return null;
70              }
71          }
72      }
73  PARSER_END(Calculadora)
```

```
74 SKIP : {
75         " "
76     |   "\t"
77     |   "\r"
78 }
79 TOKEN [IGNORE_CASE] :
80 {
81         <IMPRIMIR: "PRINT">
82     |   <A_CADENA: "A_CADENA">
83     |   <A_ENTERO: "A_ENTERO">
84     |   <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
85     |   <NUMERO: (["0"-"9"])+>
86     |   <CADENA: "\""(~["\""])*"\"">
87     |   <RETORNODECARRO: "\n">
88 }
89 /*
90 gramatica ::= ( sentFinalizada )*
91 */
92 void gramatica():{}{
93     (sentFinalizada())*
94 }
95 /*
96 sentFinalizada ::= IMPRIMIR expr '\n' | ID ASIG expr '\n' | error '\n'
97 */
98 void sentFinalizada():{
99     Simbolo s;
100    Object e;
101 }{ try {
102        <IMPRIMIR> e=expr() <RETORNODECARRO>  { usarIMPRIMIR(e); }
103    |   s=id() ":=" e=expr() <RETORNODECARRO>      { usarASIG(s, e); }
104    }catch(ParseException x){
105        System.out.println(x.toString());
106        Token t;
107        do {
108            t = getNextToken();
109        } while (t.kind != RETORNODECARRO);
110    }
111 }
112 /*
113 expr ::= term ('+' term)*
114 */
115 Object expr():{
116    Object t1, t2;
117 }{
118    t1=term() ( "+" t2=term() { t1=usarMAS(t1, t2); } )* { return t1; }
119 }
120 /*
121 term ::= fact ('*' fact)*
122 */
```

```
123 Object term():{
124     Object f1, f2;
125 }{
126     f1=fact() ( "*" f2=fact() { f1=usarPOR(f1, f2); } )*     { return f1; }
127 }
128 /*
129 fact ::= ID | NUMERO | CADENA | A_CADENA '(' expr ')' | A_ENTERO '(' expr ')'
130 */
131 Object fact():{
132     Simbolo s;
133     int i;
134     String c;
135     Object e;
136 }{
137         s=id()          { return usarID(s); }
138     |   i=numero()      { return new Integer(i); }
139     |   c=cadena()      { return c; }
140     |   <A_CADENA> "(" e=expr() ")"   { return usarA_CADENA(e); }
141     |   <A_ENTERO> "(" e=expr() ")"   { return usarA_ENTERO(e); }
142 }
143 Simbolo id():{}{
144     <ID>    {
145                 Simbolo s;
146                 if ((s = tabla.buscar(token.image)) == null)
147                     s = tabla.insertar(token.image);
148                 return s;
149             }
150 }
151 int numero():{}{
152     <NUMERO> { return Integer.parseInt(token.image); }
153 }
154 String cadena():{}{
155     <CADENA> { return token.image.substring(1, token.image.length() - 1); }
156 }
157 SKIP : {
158     <ILEGAL: (~[])>    { System.out.println("Carácter: "+image+" no esperado.");}
159 }
```

7.4 Gestión de tipos complejos

Una vez estudiados los tipos primitivos, en este apartado se estudiarán las características sintácticas y semánticas de un compilador que permite al programador construir tipos de datos complejos, punteros, *arrays*, etc.

Cuando un lenguaje de programación permite usar tipos complejos, debe suministrar construcciones sintácticas tanto para construir nuevos tipos como para referenciarlos. Algunos ejemplos de **constructores de tipo** son:

- POINTER TO. Para crear punteros.
- RECORD OF. Para crear registros.

- ARRAY OF. Para crear tablas y matrices.
- PROCEDURE RETURNS. Para crear funciones que devuelven un resultado.

Éstos no son realmente tipos sino constructores de tipo ya que, aunque permiten construir tipos complejos, no tienen sentido por sí solos y deben ser aplicados sobre un tipo base que puede ser, a su vez, otro constructor de tipo o bien un tipo primitivo. Los constructores de tipo son, por tanto, recursivos. El caso del registro es más especial porque puede aplicarse al producto cartesiano de varios tipos.

Todo esto nos lleva a dos diferencias fundamentales con respecto a la gestión de tipos primitivos que se hizo en el apartado anterior. En primer lugar, el lenguaje a definir requiere una zona de declaraciones donde enumerar las variables de un programa e indicar su tipo. En otras palabras, el tipo de una variable ya no vendrá dado por su primera asignación sino que éste deberá declararse explícitamente. Y en segundo lugar, dado que un tipo complejo puede ser arbitrariamente largo, éste no podrá codificarse con una sola letra (‘e’ para los enteros, ‘c’ para las cadenas, etc.) sino que habrá que recurrir a algún otro método.

Por otro lado, cuando se usa una variable declarada de cualquiera de los tipos anteriores, es necesario utilizar un **modificador de tipo** para referenciar los componentes. Por ejemplo:

- ^. Colocado al lado de un puntero, representa aquéllo a lo que se apunta.
- .campo. Colocado al lado de un registro, representa uno de sus campos.
- [nº]. Colocado al lado de un array, representa uno de sus elementos.
- (). Colocado al lado de una función, invoca a ésta y representa el valor de retono.

Como puede observarse, cada constructor de tipo tiene su propio modificador de tipo y éstos no son intercambiables, es decir, el “^” sólo puede aplicarse a punteros y a nada más, el “.” sólo puede aplicarse a registros, etc. Esto nos da una tabla que relaciona biunívocamente cada constructor de tipo con su modificador:

Constructores de tipos	Modificadores de tipos
POINTER TO	^
RECORD OF	.campo
ARRAY OF	[nº]
PROCEDURE RETURNS	()

También es importante darse cuenta de que un modificador de tipo no tiene porqué aplicarse exclusivamente a una variable, sino que puede aplicarse a una expresión que tenga, a su vez, otro modificador de tipo, p.ej.: “ptr^.nombre” se refiere al campo **nombre** del registro apuntado por **ptr**, y el modificador “.” se ha aplicado a la expresión **ptr^** y no a una variable directamente.

7.4.1 Objetivos y propuesta de un ejemplo

En los siguientes puntos se creará la gestión de los siguientes tipos complejos:

- Punteros
- Arrays de dimensiones desconocidas.
- Funciones sin parámetros.

El hecho de desconocer las dimensiones de los *arrays* y de obviar los parámetros de las funciones nos permitirá centrarnos exclusivamente en la gestión de tipos. Por otro lado, admitiremos los tipos básicos:

- Lógico o booleano.
- Entero.
- Real.
- Carácter.

En total tenemos cuatro tipos primitivos y tres constructores de tipos. Es importante observar que por cada tipo primitivo es necesario suministrar al programador algún mecanismo con el que pueda definir constantes de cada uno de esos tipos: constantes *booleanas*, enteras, reales y de carácter.

En lo que sigue construiremos la gestión de tipos de un compilador, lo que quiere decir que dejaremos a un lado el ejemplo de la calculadora y nos centraremos en reconocer declaraciones y expresiones válidas. Para comprobar que nuestro compilador está funcionando bien, cada vez que el programador introduzca una expresión se nos deberá mostrar un mensaje donde, en lenguaje natural, se informe del tipo de dicha expresión. Lo siguiente es un ejemplo de entrada:

```
a: POINTER TO BOOLEAN;
a^;
Es un boolean.
a;
Es un puntero a un boolean.
beta : POINTER TO ARRAY[] OF POINTER TO PROCEDURE(): INTEGER;
beta^;
Es un array de un puntero a una función que devuelve un entero.
beta[2];
Esperaba un array.
beta();
Esperaba una función.
true;
Es un boolean.
```

7.4.2 Pasos de construcción

Una vez propuesto el ejemplo, es necesario definir una gramática que reconozca las distintas cláusulas, y proponer los atributos que debe tener cada terminal y no terminal para, en base a acciones semánticas, culminar con nuestro objetivo de controlar los tipos de datos compuestos.

7.4.2.1 Gramática de partida

La gramática que se va a utilizar para el reconocimiento de las sentencias del ejemplo anterior es:

```
prog    :       /* Épsilon */
        |       prog decl ';'
        |       prog expr ';'
        |       prog error ';'
        |
        ;
decl    :       ID ',' decl
        |       ID ':' tipo
        ;
tipo    :       INTEGER
        |       REAL
        |       CHAR
        |       BOOLEAN
        |       POINTER TO tipo
        |       ARRAY '[' ']' OF tipo
        |       PROCEDURE '(' ')' ':' tipo
        ;
expr    :       ID
        |       NUM
        |       NUMREAL
        |       CARACTER
        |       CTELOGICA
        |       expr '^'
        |       expr'['NUM']'
        |       expr'('')'
        ;
```

Esta gramática expresada en notación EBNF de JavaCC queda:

```
prog():{}{
        ( bloque() )*
}
bloque():{}{
                LOOKAHEAD(2)
                decl() <PUNTOYCOMA>
        |       expr() <PUNTOYCOMA>
        |       /* Gestión del error */
}
decl():{}{
                LOOKAHEAD(2)
                <ID> ":" tipo()
        |       <ID> "," decl()
}
tipo():{}{
        (       <POINTER> <TO>
        |       <ARRAY> "[ "]" <OF>
        |       <PROCEDURE> "( ")" ":"
        )* (
                <INTEGER>
        |       <REAL>
        |       <CHAR>
```

```
        |   <BOOLEAN>
        )
}
expr():{}{
        (   <ID>    (
                        "^"
                    |   "[" <NUM> "]"
                    |   "(" ")"
                    )*
        |   <NUM>
        |   <NUMREAL>
        |   <CARACTER>
        |   <CTELOGICA>
        )
}
```

en la que puede observarse que se ha utilizado recursión a derecha para facilitar el proceso de propagación de atributos, como se verá posteriormente. Además, esta gramática no permite construcciones de la forma 8^, mientras que la dada a través de reglas de producción sí; por tanto, en una se detectará como error sintáctico y en otra como error semántico de tipos.

7.4.2.2 Gestión de atributos

La gestión de los atributos de los símbolos terminales no reviste demasiada complejidad. Los *tokens* que representan constantes no necesitan atributo alguno puesto que nuestro objetivo no es procesar valores ni generar código equivalente. Por otro lado, como nuestro lenguaje define un mecanismo para hacer explícitas las declaraciones de variables, lo mejor es que el analizador léxico proporcione al sintáctico los nombres de las variables para que éste las inserte o las busque en la tabla de símbolos según éstas aparezcan en una declaración o no.

Un problema más complejo consiste en cómo identificar el tipo de una variable compleja (ya que hemos visto que no es posible hacerlo mediante una única letra). En lo que sigue, supongamos que el usuario define la siguiente variable:

```
alfa: POINTER TO ARRAY [23] OF POINTER TO PROCEDURE(): ARRAY
      [2] OF INTEGER;
```

Indiscutiblemente vamos a necesitar una tabla de símbolos donde almacenar el nombre de cada variable y su tipo. Además, el tipo hay que almacenarlo de alguna forma inteligente que luego nos sirva para detectar errores y que permita saber qué modificador de datos se puede aplicar y cuáles no. La forma en que se decida guardar el tipo de una variable también nos debe servir para guardar el tipo de una expresión una vez se haya aplicado a una variable una secuencia de modificadores de tipo válidos.

La solución va a consistir en codificar cada tipo o constructor de tipo con una letra, incluyendo el tipo indefinido que representaremos con una 'u' de *undefined*:

Tipos y constructores	Letra-código
POINTER TO	p
ARRAY [] OF	a
PROCEDURE() :	f
BOOLEAN	b
INTEGER	i
REAL	r
CHAR	c
INDEFINIDO	u

Y un tipo completo se representará mediante una pila en la que se guardará una secuencia de estas letras. La figura 7.9 muestra las letras asociadas al tipo de la

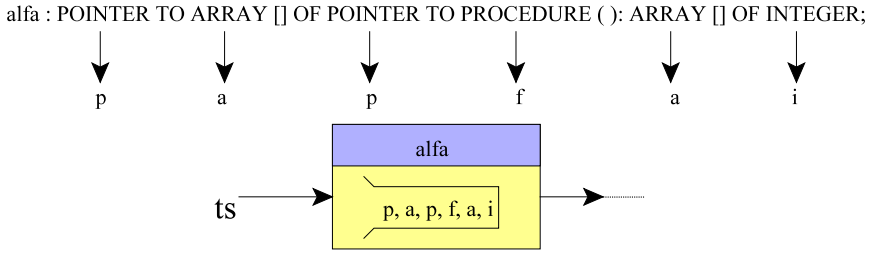


Figura 7.9. Codificación de un tipo complejo y estructura de la tabla de símbolos

variable **alfa** y la estructura de esta variable en la tabla de símbolos.

La pila se ha construido mirando el tipo de derecha a izquierda, lo que asegura que en la cima de la pila se tenga el tipo principal de **alfa**: **alfa** es, ante todo, un puntero (representado por la 'p'). Es por este motivo que en el punto se construyó una gramática recursiva a la derecha ya que las reducciones se harán en el orden en que se deben introducir elementos en la pila.

Además, si una expresión tiene asociado como atributo una pila de tipos resulta muy sencillo averiguar qué modificador se le puede aplicar:

- ^, si la cima es 'p'.
- [NUM], si la cima es 'a'.
- (), si la cima es 'f'.

Y no sólo eso resulta fácil, también resulta fácil calcular la pila de tipos de una expresión tras aplicarle un modificador de tipo. Por ejemplo, si la variable **gamma** es de tipo POINTER TO **X** entonces **gamma^** es de tipo **X**, esto es, basta con quitar la cima de la pila de tipos asociada a **gamma** para obtener la pila de tipos de la expresión **gamma^**.

7.4.2.3 Implementación de una pila de tipos

La pila de caracteres que se propone puede implementarse de muy diversas maneras. En nuestro caso, un *array* hará las veces de pila de forma que la posición 0 almacenará el número de elementos y éstos se colocarán de la posición 1 en adelante, correspondiendo la posición más alta a la cima de la pila. Con 20 posiciones será suficiente. La estructura de la pila de tipos de **alfa** se muestra en la figura 7.10.

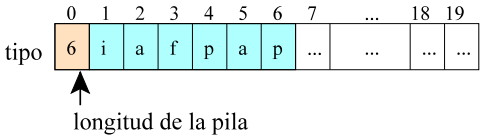


Figura 7.10. Estructura de almacenamiento del tipo de la variable **alfa**

Las operaciones de apilar, desapilar y cima son muy sencillas con esta estructura. Supongamos que la variable **tipo** es **char[20]**, entonces:

- **apilar(char x)**. Consiste en incrementar tipo[0] y luego meter x en tipo[tipo[0]]; en una sola instrucción: tipo[++tipo[0]] = x;
- **desapilar()**. Consiste únicamente en decrementar la longitud de tipo[0], quedando tipo[0]--;
- **char cima()**. La posición 0 nos da la posición en que se encuentra la cima, luego en una sola instrucción: return tipo[tipo[0]];

Aunque éste será el método que usaremos, es evidente que se está desaprovechando memoria al almacenar los tipos. Ello no debe preocuparnos dadas las cantidades de memoria que se manejan hoy día. En caso de que la memoria fuese un problema habría que utilizar algún tipo de codificación basada en bits. Dado que en total tenemos 8 letras que representar, nos bastaría con 3 bits para representar cada una de ellas, como se muestra en la siguiente tabla:

Tipos y constructores	Código de bits
POINTER TO	p - 100
ARRAY [] OF	a - 101
PROCEDURE() :	f - 110
BOOLEAN	b - 111
INTEGER	i - 001
REAL	r - 010
CHAR	c - 011
INDEFINIDO	u - 000

Además, suponiendo que el espacio de almacenamiento sobrante se rellena con 0s y dado que el código **000** se ha asociado al tipo indefinido, es posible no almacenar el tamaño de la pila sino que para encontrar la cima bastará con buscar la primera secuencia de tres bits diferente de 000. Si esta no existe es que el tipo es

indefinido.

Según esta codificación, los códigos **111**, **001**, **010**, **011** y **000** sólo pueden encontrarse en la base de la pila; es más, los códigos **100**, **101** y **110** nunca podrán estar en la base dado que requieren otro tipo por debajo sobre el que aplicarse. Este hecho nos permite utilizar los mismos códigos para representar tanto tipos primitivos como constructores de tipo: si se halla en la base de la pila es que se refiere a un tipo primitivo y en caso contrario lo que se referencia es un constructor. Si además representamos los tipos mediante un puntero a entero, el tipo indefinido vendría dado por un puntero a **null**, con lo que la codificación sería:

Tipos y constructores	Código de bits
POINTER TO	p - 01
ARRAY [] OF	a - 10
PROCEDURE() :	f - 11
BOOLEAN	b - 00
INTEGER	i - 01
REAL	r - 10
CHAR	c - 11

donde las líneas con el mismo color tienen asociado el mismo código de bits. Nótese cómo, de nuevo, no es posible usar el código **00** para representar a un constructor de tipo siempre que no se quiera guardar la longitud del tipo y los bits no utilizados se rellenen a 0.

No puede existir confusión, ya que lo que está en la base de la pila sólo puede ser un tipo primitivo; en la figura 7.11 se muestra cómo quedaría el tipo de **alfa** codificado con un entero de 16 bits.

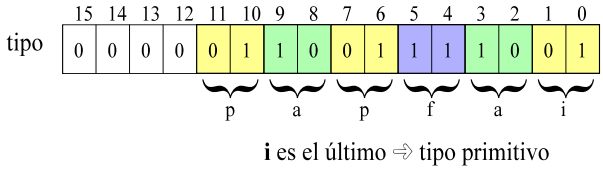


Figura 7.11. Codificación en base a bits

7.4.2.4 Acciones semánticas

Para las acciones semánticas vamos a centrarnos en la gramática dada a través de reglas de producción. En el lenguaje que estamos definiendo, existe zona de declaraciones y zona de utilización de variables (aunque pueden entremezclarse) por lo tanto es el analizador sintáctico quien inserta los identificadores en la tabla de símbolos en el momento en que se declaran.

Como puede observarse en las reglas de **prog**, existe una regla épsilon que nos

permite entrar por ella justo al comienzo del análisis, como paso base de la recursión de las reglas de **prog** sobre sí misma.

En las reglas de **decl** y de **tipo**, hacemos la gramática recursiva a la derecha. Así, las reglas de **tipo** permiten construir la pila desde la base hasta la cima, ejecutando tan sólo operaciones **apilar**. Por otro lado, las reglas **decl** permiten asignar el tipo a cada uno de los identificadores declarados, también de derecha a izquierda, tal y como se ilustra en la figura 7.12. Como puede apreciarse en ella, cuando el analizador sintáctico se encuentra un tipo primitivo construye una pila con un sólo carácter (el correspondiente al tipo encontrado) y lo asigna como atributo del no terminal **tipo**. A medida que se reduce a nuevos no terminales **tipo** en base a las reglas recursivas a la derecha, se toma el atributo del **tipo** del consecuente como punto de partida para construir el atributo del antecedente; basta con añadir a éste el carácter correspondiente al constructor de tipo de que se trate (marcados en gris en la figura 7.12). Una vez construido el tipo completo, el analizador sintáctico debe reducir por la regla:

```
decl   :     ID ':' tipo
```

de tal manera que debe crearse una entrada en la tabla de símbolos para la variable indicada como atributo de **ID** (**c** según la figura 7.12); el tipo de esta variable vendrá dado por el atributo de **tipo**. Por supuesto es necesario comprobar que no se ha realizado una redeclaración de variables. Dado que es posible realizar declaraciones de varias variables simultáneamente, el resto de variables (**b** y **a** segun la figura 7.12) se reconocen mediante la reducción de la regla:

```
decl   :     ID ',' decl
```

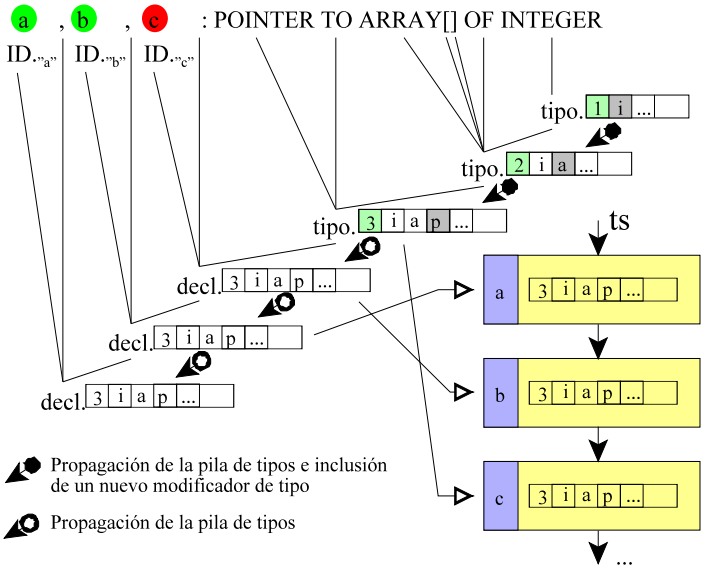


Figura 7.12. Construcción de una pila-tipo y asignación a cada variable en la tabla de símbolos

Obviamente el proceso debe ser muy similar al de la regla anterior, esto es, crear una entrada en la tabla de símbolos para la variable del atributo de **ID** y asignarle una pila de tipos. Claro está, para ello es necesario disponer de la pila de tipos como atributo del no terminal **decl**, por lo que dicha pila debe ser propagada en ambas reglas de producción desde el no terminal del consecuente al antecedente, o sea, a **decl**. En conclusión, tanto **tipo** como **decl** tienen como atributo una pila de tipos.

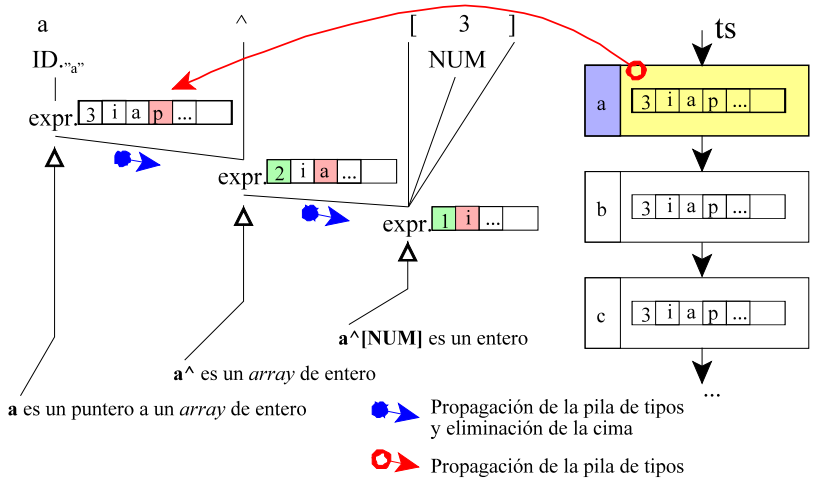
Pasemos ahora a estudiar las reglas de las expresiones. El objetivo es que una expresión tenga también asociada una pila de tipos de manera que, una vez completado el análisis de la expresión completa, su pila de tipos se pase como parámetro a una función que traslade su contenido a una frase en lenguaje natural.

Las expresiones pueden dividirse en tres bloques: 1) primitivas, formadas por una constante de cualquier tipo; 2) simples, formadas por una variable; y 3) complejas, formadas por una expresión simple con modificadores de tipo a su derecha.

Las expresiones del tipo 1 tienen asociada una pila de tipos con un único carácter, que se corresponderá con el tipo del literal reconocido:

- 'i' para NUM.
- 'r' para NUMREAL.
- 'c' para CARACTER.
- 'b' para CTELOGICA.

Las expresiones del tipo 2 tienen una pila de tipos exactamente igual que la de la variable a que representan. Si la variable no ha sido declarada entonces la pila de tipos tendrá únicamente una 'u' que representa al tipo indefinido (no se puede usar la letra 'i' porque se confundiría con la 'i' de **INTEGER**).



Las expresiones del tipo 3 parten de una expresión (que tiene una pila de tipos asociada) junto con un modificador de tipo. En este caso hay que controlar que el tipo principal de la expresión (representado por la cima de su pila de tipos), es compatible con el modificador de tipo, tal y como se vio al final del apartado 7.4.2.2. Por ejemplo, si encontramos "expr^", "expr[NUM]" o "expr()" hay que controlar que el tipo principal de alfa sea 'p', 'a' o 'f' respectivamente. Suponiendo que la variable **a** ha sido declarada tal y como aparece en la figura 7.12, la figura muestra cómo varía la pila de tipos durante el reconocimiento de la expresión "a^[3]".

En el momento en que reduce por la regla:

```
expr    :    ID
```

se busca en la tabla de símbolos a la variable que viene dada como atributo de **ID**; una vez encontrada, el atributo de **expr** será una copia de la pila de tipos de dicha variable. A medida que se van aplicando las reglas:

```
expr    :    expr '^'
        |    expr'['NUM']'
        |    expr'('')'
        ;
```

se va comprobando que el modificador de la regla aplicada sea compatible con el tipo de la cima de la pila de la expresión a que se aplica. Si es compatible, entonces el tipo de la expresión resultante (el antecedente) será el mismo que la del consecuente pero sin la cima. Si no se debe emitir un mensaje de error y la expresión resultante será de tipo indefinido.

En resumen, los constructores de tipo construyen la pila metiendo caracteres en ella, mientras que los modificadores de tipo destruyen la pila sacando caracteres por la cima.

7.4.3 Solución con Lex/Yacc

Lo más interesante de esta solución radica en la implementación del tipo de datos pila y de sus operaciones, especialmente de los controles de los límites del *array* que se han incluido. Estas operaciones obedecen básicamente al esquema propuesto en el apartado 7.4.2.3, y se codifican entre las líneas 9 y 32 del fichero **TabSimb.c** que se presenta a continuación:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define TAM_PILA 21
4  #define TAM_NOMBRE 20
5  /*
6       El caracter 0 indica el número de posiciones ocupadas en
7       el resto de la cadena
8  */
9  typedef char pila_tipo[TAM_PILA];
10 void crearPila(pila_tipo tipo, char x){ tipo[0]=1; tipo[1]=x; tipo[TAM_PILA-1]=0; }
11 void insertarPila(pila_tipo tipo, char x){
12      if (tipo[0] < TAM_PILA-2) tipo[++tipo[0]]=x;
```

```
13      else printf("Tipo demasiado complejo.\n");
14  }
15  void eliminarPila(pila_tipo tipo){ if (tipo[0] > 0) tipo[0]--; }
16  char cimaPila(pila_tipo tipo) { return tipo[tipo[0]]; }
17  void copiarPila(pila_tipo destino, pila_tipo origen) { strcpy(destino, origen); }
18  void verPila(pila_tipo tipo) {
19      unsigned char cont;
20      printf("El tipo es ");
21      for(cont = tipo[0]; cont>0; cont--)
22      switch(tipo[cont]) {
23          case('i'): { printf("un entero.\n"); break; }
24          case('r'): { printf("un real.\n"); break; }
25          case('b'): { printf("un booleano.\n"); break; }
26          case('c'): { printf("un caracter.\n"); break; }
27          case('p'): { printf("un puntero a "); break; }
28          case('a'): { printf("un array de "); break; }
29          case('f'): { printf("una funcion que devuelve "); break; }
30          case('u'): { printf("indefinido.\n"); break; }
31      };
32  }
33  /* Definición de la tabla de símbolos */
34  typedef struct _simbolo {
35      struct _simbolo * sig;
36      char nombre[TAM_NOMBRE];
37      pila_tipo tipo;
38  } simbolo;
39  void insertar(simbolo ** p_t, char nombre[TAM_NOMBRE], pila_tipo tipo) {
40      simbolo * s = (simbolo *) malloc(sizeof(simbolo));
41      strcpy(s->nombre, nombre);
42      strcpy(s->tipo, tipo);
43      s->sig = (*p_t);
44      (*p_t) = s;
45  }
46  simbolo * buscar(simbolo * t, char nombre[TAM_NOMBRE]) {
47      while ( (t != NULL) && (strcmp(nombre, t->nombre)) )
48          t = t->sig;
49      return (t);
50  };
51  void imprimir(simbolo * t) {
52      while (t != NULL) {
53              printf("%s. ", t->nombre);
54              verPila(t->tipo);
55              t = t->sig;
56          }
57  }
```

Como se deduce de la función **verPila()**, convertir una pila de tipos en una frase en lenguaje natural que la describa es tan sencillo como concatenar trozos de texto prefijados para cada letra o código de tipo.

El fichero **TipCompl.lex** realiza el análisis léxico, prestando cuidado de que el programador no escriba identificadores demasiado largos. En caso de que un identificador tenga más de 19 caracteres, el propio analizador lo trunca informando de ello en un mensaje por pantalla. Es muy importante informar de este truncamiento, ya que puede haber identificadores muy largos en los que coincidan sus primeros 19 caracteres, en cuyo caso nuestro compilador entenderá que se tratan del mismo identificador. El programa Lex es:

```
%%
[0-9]+          {return NUM;}
[0-9]+\.[0-9]+ {return NUMREAL;}
\'.\'           {return CARACTER;}
"TRUE"  |
"FALSE"         {return CTELOGICA;}
"INTEGER"       {return INTEGER;}
"REAL"          {return REAL;}
"CHAR"          {return CHAR;}
"BOOLEAN"       {return BOOLEAN;}
"POINTER"       {return POINTER;}
"TO"            {return TO;}
"ARRAY"         {return ARRAY;}
"OF"            {return OF;}
"PROCEDURE"     {return   PROCEDURE;}
[a-zA-Z_][a-zA-Z0-9_]*      {
                            if (strlen(yytext) >= TAM_NOMBRE) {
                                printf("Variable %s truncada a ", yytext);
                                yytext[TAM_NOMBRE-1] = 0;
                                printf("%s.\n", yytext);
                            }
                            strcpy(yylval.nombre, yytext);
                            return ID;
                        }
[ \t\n]+        {;}
.               {return yytext[0];}
```

El programa Yacc que soluciona el analizador sintáctico y semántico se mantiene relativamente reducido al haberse extraído la funcionalidad de la pila de tipos y de la tabla de símbolos en funciones aparte en el fichero **TabSimb.c**. El fichero **TipCompy.yac** es:

```
%{
#include "TabSimb.c"
simbolo * tabla=NULL;
%}
%union {
      pila_tipo tipo;
      char nombre[20];
}
%token INTEGER REAL CHAR BOOLEAN POINTER TO
%token ARRAY OF PROCEDURE
```

```
11 %token NUM CARACTER NUMREAL CTELOGICA
12 %token <nombre> ID
13 %type <tipo> tipo expr decl
14 %start prog
15 %%
16 prog    :    /*´Epsilon */
17          |    prog decl ';'        {verPila($2);}
18          |    prog expr ';'        {verPila($2);}
19          |    prog error ';'       {yyerrok;}
20          ;
21 decl    :    ID ',' decl {
22                                        copiarPila($$,$3);
23                                        if(buscar(tabla, $1)!= NULL)
24                                             printf("%s redeclarada.\n",$1);
25                                        else
26                                             insertar(&tabla, $1, $3);
27                                   }
28          | ID ':' tipo {
29                                        copiarPila($$,$3);
30                                        if(buscar(tabla, $1)!= NULL)
31                                             printf("%s redeclarada.\n",$1);
32                                        else
33                                             insertar(&tabla, $1, $3);
34                                   }
35      ;
36 tipo  :        INTEGER    { crearPila($$, 'i'); }
37          |    REAL       { crearPila($$, 'r'); }
38          |    CHAR       { crearPila($$, 'c'); }
39          |    BOOLEAN    { crearPila($$, 'b'); }
40          |    POINTER TO tipo {
41                                   copiarPila($$,$3);
42                                   insertarPila($$, 'p');
43                              }
44          |    ARRAY '['  ']' OF tipo {
45                                   copiarPila($$,$5);
46                                   insertarPila($$, 'a');
47                              }
48          |    PROCEDURE '("):'tipo {
49                                   copiarPila($$,$5);
50                                   insertarPila($$, 'f');
51                              }
52          ;
53 expr    :    ID                 {
54                                        if(buscar(tabla, $1)==NULL) {
55                                             crearPila($$, 'u');
56                                             printf("%s no declarada.\n",$1);
57                                        } else
58                                             copiarPila($$,buscar(tabla,$1)->tipo);
59                                   }
```

```
60          |   NUM             { crearPila($$, 'i'); }
61          |   NUMREAL        { crearPila($$, 'r'); }
62          |   CARACTER       { crearPila($$, 'c'); }
63          |   CTELOGICA      { crearPila($$, 'b'); }
64          |   expr '^'       {
65                                  if(cimaPila($1) != 'p') {
66                                      crearPila($$, 'u');
67                                      printf("Esperaba un puntero.\n");
68                                  } else {
69                                      copiarPila($$, $1);
70                                      eliminarPila($$);
71                                  }
72                              }
73          |   expr'['NUM']'  {
74                                  if(cimaPila($1) != 'a') {
75                                      crearPila($$, 'u');
76                                      printf("Esperaba un array.\n");
77                                  } else {
78                                      copiarPila($$, $1);
79                                      eliminarPila($$);
80                                  }
81                              }
82          |   expr'(' ')'    {
83                                  if(cimaPila($1) != 'f') {
84                                      crearPila($$, 'u');
85                                      printf("Esperaba una funcion.\n");
86                                  } else {
87                                      copiarPila($$, $1);
88                                      eliminarPila($$);
89                                  }
90                              }
91          ;
92  %%
93  #include "TipCompl.c"
94  void main() {
95      yyparse();
96      imprimir(tabla);
97  }
98  void yyerror(char * s) {
99      printf("%s\n",s);
100 }
```

7.4.4 Solución con JFlex/Cup

La solución con Java pasa por crear un símbolo formado por un nombre de tipo **String** y una pila de caracteres de tipo **Stack**. Además, lo más sensato es introducir en esta misma clase la función que genera la cadena de texto que describe en lenguaje natural la pila de tipos. La clase **Simbolo** queda:

```
1  class Simbolo{
```

```
2       String nombre;
3       Stack tipo;
4       public Simbolo(String nombre, Stack tipo){
5           this.nombre = nombre;
6           this.tipo = tipo;
7       }
8       public static String tipoToString(Stack st){
9           String retorno = "";
10          for(int cont = st.size()-1; cont>=0; cont--)
11          switch(((Character)st.elementAt(cont)).charValue()) {
12              case('i'): { retorno += "un entero."; break; }
13              case('r'): { retorno += "un real."; break; }
14              case('b'): { retorno += "un booleano."; break; }
15              case('c'): { retorno += "un caracter."; break; }
16              case('p'): { retorno += "un puntero a "; break; }
17              case('a'): { retorno += "un array de "; break; }
18              case('f'): { retorno += "una funcion que devuelve "; break; }
19              case('u'): { retorno += "indefinido."; break; }
20          };
21          return retorno;
22      }
23  }
```

La tabla de símbolos es muy parecida a la de ejercicios anteriores, excepto por el hecho de que la función **insertar()** también acepta un objeto de tipo **Stack**. Esto se debe a que en el momento de la inserción de un símbolo también se conoce su tipo, dado que es Cup quien realiza las inserciones. Así, la clase **TablaSimbolos** queda:

```
1   import java.util.*;
2   public class TablaSimbolos{
3       HashMap t;
4       public TablaSimbolos(){
5           t = new HashMap();
6       }
7       public Simbolo insertar(String nombre, Stack st){
8           Simbolo s = new Simbolo(nombre, st);
9           t.put(nombre, s);
10          return s;
11      }
12      public Simbolo buscar(String nombre){
13          return (Simbolo)(t.get(nombre));
14      }
15      public void imprimir(){
16          Iterator it = t.values().iterator();
17          while(it.hasNext()){
18              Simbolo s = (Simbolo)it.next();
19              System.out.println(s.nombre + ": " + Simbolo.tipoToString(s.tipo));
20          }
21      }
22  }
```

El programa en JFlex se limita a reconocer los *tokens* necesarios, de manera análoga a como se hacía en Lex. Lo único interesante viene dado por las líneas 5 y 6 del siguiente código; la declaración de las líneas 10 y 11 hace que en la clase **Yylex** se creen variables que llevan la cuenta del número de línea y de columna donde comienza el *token* actual. Dado que estas variables son privadas es necesario crear las funciones de las líneas para exteriorizarlas. El código de **TipCompl.jflex** es:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  %%
4  %{
5        public int linea(){ return yyline+1; }
6        public int columna(){ return yycolumn+1; }
7  %}
8  %unicode
9  %cup
10 %line
11 %column
12 %%
13 [:digit:]+                    { return new Symbol(sym.NUM); }
14 [:digit:]+\.[:digit:]+ { return new Symbol(sym.NUMREAL); }
15 \'.\'                          { return new Symbol(sym.CARACTER); }
16 "TRUE"  |
17 "FALSE"                       { return new Symbol(sym.CTELOGICA); }
18 "INTEGER"          { return new Symbol(sym.INTEGER); }
19 "REAL"             { return new Symbol(sym.REAL); }
20 "CHAR"             { return new Symbol(sym.CHAR); }
21 "BOOLEAN"          { return new Symbol(sym.BOOLEAN); }
22 "POINTER"          { return new Symbol(sym.POINTER); }
23 "TO"               { return new Symbol(sym.TO); }
24 "ARRAY"            { return new Symbol(sym.ARRAY); }
25 "OF"               { return new Symbol(sym.OF); }
26 "PROCEDURE"   { return   new Symbol(sym.PROCEDURE); }
27 "^"   { return new Symbol(sym.CIRCUN); }
28 "["   { return new Symbol(sym.LCORCH); }
29 "]"   { return new Symbol(sym.RCORCH); }
30 "("   { return new Symbol(sym.LPAREN); }
31 ")"   { return new Symbol(sym.RPAREN); }
32 ":"   { return new Symbol(sym.DOSPUN); }
33 ";"   { return new Symbol(sym.PUNTOYCOMA); }
34 ","   { return new Symbol(sym.COMA); }
35 [:jletter:][:jletterdigit:]*  { return new Symbol(sym.ID, yytext()); }
36 [ \t\n\r]+          {;}
37 .            { System.out.println("Error léxico: '"+yytext()+"'."); }
```

Por último, el programa en Cup realiza algunas declaraciones importantes en el área de código del analizador sintáctico. En concreto, crea como estática la tabla de símbolos, de manera que sea accesible desde las acciones semánticas mediante la referencia **parser.tabla**. También se crea un analizador léxico **miAnalizadorLexico**

de tipo **Yylex** como variable de instancia; esto se hace con el objetivo de poder referenciar las funciones **linea()** y **columna()** del analizador léxico en un mensaje de error personalizado. Los mensajes de error se pueden personalizar reescribiendo la función **syntax_error()** que toma como parámetro el *token* actual que ha provocado el error. En nuestro caso se ha reescrito esta función en las líneas 20 a 23. En éstas se llama, a su vez, a la función **report_error()** que se encarga de sacar un mensaje por la salida de error (**System.err**) y que, en caso necesario, también puede ser reescrita por el desarrollador. De hecho, la salida de error se ha utilizado en todos los mensajes de error semántico que se visualizan: "variable no declarada", "variable redeclarada", "esperaba un puntero", etc.

Por otro lado, y como ya se ha indicado, se ha utilizado un objeto de tipo **Stack** para almacenar la pila de tipos. Nótese cómo, por regla general, se utiliza compartición estructural, esto es, no se generan copias de la pila de tipos. Por ejemplo, durante una declaración, la pila de tipos se construye al encontrar un tipo primitivo y a medida que se encuentran constructores de tipos se van añadiendo elementos a la pila original, sin cear copias de la misma (se usa asignación pura y no la función **clone()**). El único caso en el que es necesario trabajar con una copia de una pila de tipos es cuando se recupera el tipo de un identificador como expresión base sobre la que aplicar modificadores posteriores (véase la línea 82); si no se hiciera una copia **clone()** en este caso, cada vez que se aplicara un modificador de tipo se estaría cambiando el tipo del identificador (y de todos los que se declararon junto a él).

El fichero **TipCompl.cup** queda:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  import java.util.*;
4  parser code {:
5       static TablaSimbolos tabla = new TablaSimbolos();
6       Yylex miAnalizadorLexico;
7       public static void main(String[] arg){
8            parser parserObj = new parser();
9            parserObj.miAnalizadorLexico =
10                                new Yylex(new InputStreamReader(System.in));
11           parserObj.setScanner(parserObj.miAnalizadorLexico);
12           try{
13                parserObj.parse();
14                tabla.imprimir();
15           }catch(Exception x){
16                x.printStackTrace();
17                System.err.println("Error fatal.");
18           }
19      }
20      public void syntax_error(Symbol cur_token){
21           report_error("Error de sintaxis: linea "+miAnalizadorLexico.linea()+
22                                     ", columna "+miAnalizadorLexico.columna(), null);
23      }
```

```
24 :};
25 terminal INTEGER, REAL, CHAR, BOOLEAN, POINTER, TO;
26 terminal ARRAY, OF, PROCEDURE;
27 terminal NUM, CARACTER, NUMREAL, CTELOGICA;
28 terminal RPAREN, LPAREN, RCORCH, LCORCH, CIRCUN;
29 terminal DOSPUN, PUNTOYCOMA, COMA;
30 terminal String ID;
31 non terminal Stack tipo, expr, decl;
32 non terminal prog;
33 /* Gramática */
34 start with prog;
35 prog      ::=  /*´Epsilon */
36           |          prog decl:d PUNTOYCOMA
37                                    {: System.out.println(Simbolo.tipoToString(d)); :}
38           |          prog expr:e PUNTOYCOMA
39                                    {: System.out.println(Simbolo.tipoToString(e)); :}
40           |          prog error PUNTOYCOMA
41                                    {: ; :}
42           ;
43 decl      ::=  ID:id COMA decl:t {:
44                                    RESULT = t;
45                                    if(parser.tabla.buscar(id)!= null)
46                                         System.err.println(id + " redeclarada.");
47                                    else
48                                         parser.tabla.insertar(id, t);
49                               :}
50           |    ID:id DOSPUN tipo:t {:
51                                    RESULT = t;
52                                    if(parser.tabla.buscar(id)!= null)
53                                         System.err.println(id + " redeclarada.");
54                                    else
55                                         parser.tabla.insertar(id, t);
56                               :}
57      ;
58 tipo ::=  INTEGER    {: RESULT = new Stack(); RESULT.push(new Character('i')); :}
59      |    REAL       {: RESULT = new Stack(); RESULT.push(new Character('r')); :}
60      |    CHAR       {: RESULT =new Stack(); RESULT.push(new Character('c')); :}
61      |    BOOLEAN    {: RESULT =new Stack(); RESULT.push(new Character('b')); :}
62      |    POINTER TO tipo:t    {:
63                                    RESULT = t;
64                                    RESULT.push(new Character('p'));
65                               :}
66      |    ARRAY LCORCH RCORCH OF tipo:t {:
67                                    RESULT = t;
68                                    RESULT.push(new Character('a'));
69                               :}
70      |    PROCEDURE RPAREN LPAREN DOSPUN tipo:t {:
71                                    RESULT = t;
72                                    RESULT.push(new Character('f'));
```

```
                                        :}
    ;
expr    ::= ID:id       {:
                    Simbolo s;
                    if ((s = parser.tabla.buscar(id)) == null) {
                        RESULT = new Stack();
                        RESULT.push(new Character('u'));
                        System.err.println(id + " no declarada.");
                    } else
                        RESULT = (Stack)s.tipo.clone();
                :}
    | NUM       {: RESULT = new Stack(); RESULT.push(new Character('i')); :}
    | NUMREAL   {: RESULT = new Stack(); RESULT.push(new Character('r')); :}
    | CARACTER  {: RESULT =new Stack(); RESULT.push(new Character('c')); :}
    | CTELOGICA{: RESULT = new Stack(); RESULT.push(new Character('b')); :}
    | expr:t CIRCUN     {:
                if(((Character)t.peek()).charValue() != 'p') {
                    RESULT = new Stack(); RESULT.push(new Character('u'));
                    if(((Character)t.peek()).charValue() != 'u')
                            System.err.println("Esperaba un puntero.");
                } else {
                    RESULT = t;
                    RESULT.pop();
                }
            :}
    | expr:t LCORCH NUM RCORCH {:
                if(((Character)t.peek()).charValue() != 'a') {
                    RESULT = new Stack(); RESULT.push(new Character('u'));
                    if(((Character)t.peek()).charValue() != 'u')
                        System.err.println("Esperaba un array.");
                } else {
                    RESULT = t;
                    RESULT.pop();
                }
            :}
|       expr:t LPAREN RPAREN {:
                if(((Character)t.peek()).charValue() != 'p') {
                    RESULT = new Stack(); RESULT.push(new Character('u'));
                    if(((Character)t.peek()).charValue() != 'u')
                        System.err.println("Esperaba una función.");
                } else {
                    RESULT = t;
                    RESULT.pop();
                }
            :}
        ;
```

7.4.5 Solución con JavaCC

La solución con JavaCC utiliza acciones semánticas muy parecidas a las de la solución con JFlex y Cup. De hecho, las clases **TablaSimbolos** y **Simbolo** son exactamente iguales. Por otro lado, la pila de tipos se construye invertida en la regla de **tipo**, por lo que es necesario darle la vuelta antes de devolverla en las líneas 117 a 121.

Aunque se ha utilizado un objeto de tipo **Stack** para almacenar los caracteres que representan los tipos, también podría haberse usado un *array* de **char**, de la misma manera en que se hizo en la solución con Lex y Yacc. En tal caso, no habría sido necesario controlar los límites del *array* ya que, en caso de que éstos se violaran, se elevaría una excepción que se capturaría en el **try-catch** de las líneas 87 a 92 como si se tratase de cualquier error sintáctico. El fichero **TiposComplejos.jj** queda:.

```
1  PARSER_BEGIN(TiposComplejos)
2      import java.util.*;
3      public class TiposComplejos{
4          static TablaSimbolos tabla = new TablaSimbolos();
5          public static void main(String args[]) throws ParseException {
6          new TiposComplejos(System.in).prog();
7              tabla.imprimir();
8          }
9          private static Stack declaracion(String id, Stack t){
10             if(tabla.buscar(id)!= null)
11                 System.err.println(id + " redeclarada.");
12             else
13                 tabla.insertar(id, t);
14             return t;
15         }
16         private static Stack invertir(Stack in){
17             Stack out=new Stack();
18             while(!in.empty())
19                 out.push(in.pop());
20             return out;
21         }
22         private static Stack utilizacion(String id){
23             Simbolo s;
24             if ((s = tabla.buscar(id)) == null) {
25                 System.err.println(id + " no declarada.");
26                 return nuevaPila('u');
27             } else
28                 return (Stack)s.tipo.clone();
29         }
30         private static Stack nuevaPila(char c){
31             Stack st = new Stack();
32             st.push(new Character(c));
33             return st;
34         }
35         private static Stack comprobar(Stack st, String mensaje, char c){
36             if(((Character)st.peek()).charValue() != c) {
37                 if(((Character)st.peek()).charValue() != 'u')
```

```
38                          System.err.println("Esperaba "+mensaje+".");
39                    return nuevaPila('u');
40              } else {
41                    st.pop();
42                    return st;
43              }
44        }
45    }
46 PARSER_END(TiposComplejos)
47 SKIP : {
48        " "
49    |   "\t"
50    |   "\n"
51    |   "\r"
52 }
53 TOKEN [IGNORE_CASE] :
54 {
55        <POINTER: "POINTER">
56    |   <TO: "TO">
57    |   <ARRAY: "ARRAY">
58    |   <OF: "OF">
59    |   <PROCEDURE: "PROCEDURE">
60    |   <INTEGER: "INTEGER">
61    |   <REAL: "REAL">
62    |   <CHAR: "CHAR">
63    |   <BOOLEAN: "BOOLEAN">
64    |   <NUM: (["0"-"9"])+>
65    |   <NUMREAL: (["0"-"9"])+"."(["0"-"9"])+>
66    |   <CARACTER: ("\'"~["'"\'"")>
67    |   <CTELOGICA: ("TRUE"|"FALSE")>
68    |   <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
69    |   <PUNTOYCOMA: ";">
70 }
71 /*
72 prog ::= ( bloque )*
73 */
74 void prog():{}{
75      ( bloque() )*
76 }
77 /*
78 bloque ::= decl ';' | expr ';' | error ';'
79 */
80 void bloque():{
81      Stack st;
82 }{
83      try {
84           LOOKAHEAD(2)
85           st=decl() <PUNTOYCOMA> { System.out.println(Simbolo.tipoToString(st)); }
86      |    st=expr() <PUNTOYCOMA> { System.out.println(Simbolo.tipoToString(st)); }
```

```
87        }catch(ParseException x){
88            System.err.println(x.toString());
89            Token t;
90            do {
91                t = getNextToken();
92            } while (t.kind != PUNTOYCOMA);
93        }
94  }
95  /*
96  decl ::= ID ( ',' ID )* ':' tipo
97  */
98  Stack decl():{
99        String nombre;
100       Stack t;
101 }{
102           LOOKAHEAD(2)
103           nombre=id() ":" t=tipo() { return declaracion(nombre, t); }
104       |   nombre=id() "," t=decl() { return declaracion(nombre, t); }
105 }
106 /*
107 tipo ::= ( POINTER TO | ARRAY '[' ']' OF | PROCEDURE '(' ')' ':' )*
108                                          ( INTEGER | REAL | CHAR | BOOLEAN )
109 */
110 Stack tipo():{
111       Stack st = new Stack();
112 }{
113       (   <POINTER> <TO>          { st.push(new Character('p')); }
114       |   <ARRAY> "[" "]" <OF>    { st.push(new Character('a')); }
115       |   <PROCEDURE> "(" ")" ":" { st.push(new Character('f')); }
116       )* (
117           <INTEGER>               { st.push(new Character('i')); return invertir(st); }
118       |   <REAL>                  { st.push(new Character('r')); return invertir(st); }
119       |   <CHAR>                  { st.push(new Character('c')); return invertir(st); }
120       |   <BOOLEAN>               { st.push(new Character('b')); return invertir(st); }
121       )
122 }
123 /*
124 expr ::= NUM | NUMREAL | CARACTER | CTELOGICA | ID ( '^' | '[' NUM ']' | '(' ')' )*
125 */
126 Stack expr():{
127       String nombre;
128       Stack st;
129 }{
130       (   nombre=id() { st = utilizacion(nombre); }
131                   (
132                       "^"                 { st = comprobar(st, "un puntero", 'p'); }
133                   |   "[" <NUM> "]"       { st = comprobar(st, "un array", 'a'); }
134                   |   "(" ")"             { st = comprobar(st, "una función", 'f'); }
135                   )*          { return st; }
```

```
136       |     <NUM>                 { return nuevaPila('e'); }
137       |     <NUMREAL>             { return nuevaPila('r'); }
138       |     <CARACTER>            { return nuevaPila('c'); }
139       |     <CTELOGICA>          { return nuevaPila('b'); }
140       )
141   }
142   String id():{}{
143       <ID>      { return token.image; }
144   }
145   SKIP : {
146       <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
147   }
```

Capítulo 8

Generación de código

8.1 Visión general

Según el modelo de arquitectura de un compilador en el que éste se divide en *frontend* y *backend*, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto, ya sea en forma de código máquina o ensamblador. Los detalles del lenguaje objeto se confinan en la etapa final, si esto es posible, lo que facilita la reutilización del *frontend* para crear otros compiladores del mismo lenguaje pero que generan código para otras plataformas. De esta forma, aunque a priori puede resultar más fácil traducir un programa fuente directamente al lenguaje objeto, las dos ventajas principales de utilizar una forma

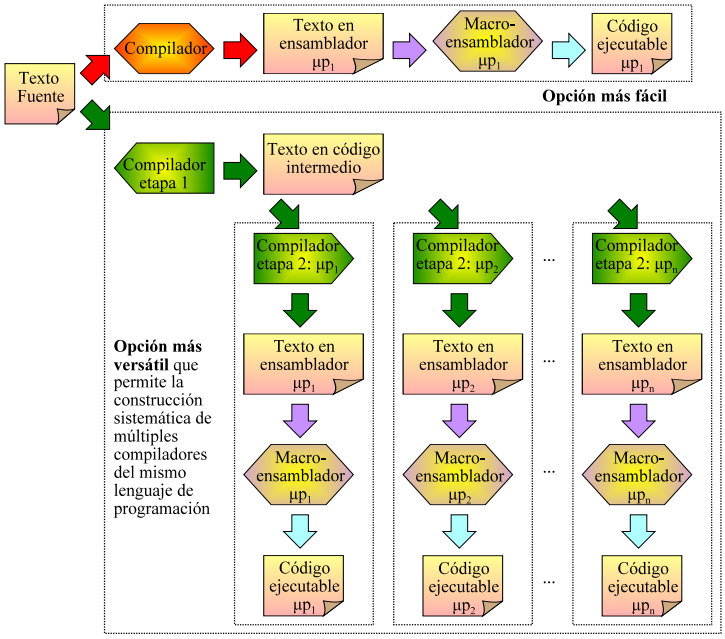


Figura 8.1. La construcción de un compilador mediante división en etapa *frontend* y etapa *backend* se realiza utilizando un código intermedio independiente de la máquina destino y en el que se codifica el programa fuente. El programa en código intermedio resultante es la salida de la etapa *frontend* y la entrada al *backend*

intermedia independiente de la máquina destino son:

- Se facilita la re-destinación, o sea, se puede crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente.
- Se puede aplicar a la representación intermedia un optimizador de código independiente de la máquina, lo que permite reutilizar también esta fase del compilador independientemente de la máquina destino.

La figura 8.1 muestra cómo el desarrollador debe optar por crear un compilador mediante un diseño versátil o mediante un diseño fácil. La elección final depende de la utilidad que se le vaya a dar al código fuente del compilador a corto o medio plazo, esto es, si se van a crear varios compiladores o uno sólo.

La filosofía versátil en la construcción de compiladores llega a su extremo en la implementación de lenguajes que son compilados y pseudointerpretados en ejecución. Esto quiere decir que en tiempo de compilación se genera un código máquina propio de un microprocesador virtual (llamado código-P en UCSD Pascal, *bytecodes* en Java, etc.) que, a su vez, se interpreta en tiempo de ejecución a través de lo que se llama **motor de ejecución**. A estos lenguajes no se les puede catalogar como interpretados ya que lo que se interpreta en tiempo de ejecución no es exactamente el programa fuente; pero tampoco se les puede considerar compilados del todo ya que lo que se genera en tiempo de compilación no es exactamente código máquina. Esta filosofía de diseño tiene la ventaja de que facilita la portabilidad de los programas. El lenguaje más actual que trabaja con esta filosofía es Java, que utiliza ficheros **.class**, en lugar de **.exe**. Los ficheros **.class** contienen *bytecodes* que se someten a una *Java Virtual Machine* (JVM) en tiempo de ejecución, para que los interprete. La JVM hace las veces de microprocesador virtual y los *bytecodes* hacen las veces de instrucciones máquina. Por supuesto, para poder ejecutar los *bytecodes* en diferentes plataformas es necesario que cada una de ellas posea una implementación adaptada de la JVM.

En este capítulo se muestra cómo se pueden utilizar los métodos de análisis dirigidos por la sintaxis para traducir un programa fuente a un programa destino equivalente escrito en código intermedio. Dado que las declaraciones de variables no generan código, los siguientes epígrafes se centran en las sentencias propias de un lenguaje de programación imperativo, especialmente asignaciones y cláusulas de control del flujo de ejecución. La generación de código intermedio se puede intercalar en el análisis sintáctico mediante las apropiadas acciones semánticas.

También es importante notar que, a veces, puede resultar interesante construir un traductor fuente-fuente en lugar de un compilador completo. Por ejemplo, si disponemos de un compilador de C y queremos construir un compilador de Pascal, puede resultar mucho más cómodo construir un programa que traduzca de Pascal a C, de manera que, una vez obtenido el programa C equivalente al de Pascal, bastará con compilar éste para obtener el resultado apetecido. Básicamente, las técnicas que se verán en este capítulo para generar código intermedio también pueden aplicarse a la generación de código de alto nivel.

8.2 Código de tercetos

Con el objetivo de facilitar la comprensión de la fase de generación de código, no nos centraremos en el código máquina puro de ningún microprocesador concreto, sino en un código intermedio cercano a cualquiera de ellos. Esta aproximación facilitará, además, la optimización del código generado.

Cada una de las instrucciones que podemos generar posee un máximo de cuatro apartados:

- Operando 1º (dirección de memoria donde se encuentra el primer operando).
- Operando 2º (dirección de memoria donde se encuentra el segundo operando).
- Operador (código de operación)
- Resultado (dirección de memoria donde albergar el resultado, o a la que saltar en caso de que se trate de una operación de salto).

Asumiremos que se permite tanto el direccionamiento directo como el inmediato, esto es, la dirección de un operando puede sustituirse por el operando en sí. A este tipo de instrucciones se las denomina códigos de 3 direcciones, tercetos, o códigos de máximo 3 operandos.

En esencia, los tercetos son muy parecidos a cualquier código ensamblador, existiendo operaciones para sumar, restar, etc. También existen instrucciones para controlar el flujo de ejecución, y pueden aparecer etiquetas simbólicas en medio del código con objeto de identificar el destino de los saltos.

No todas las instrucciones tienen porqué poseer exactamente todos los apartados mencionados. A modo introductorio, los tercetos que podemos usar son:

- Asignación binaria: **x := y op z**, donde **op** es una operación binaria aritmética o lógica. Aunque debe haber operaciones diferentes en función del tipo de datos con el que se trabaje (no es lo mismo sumar enteros que sumar reales), para facilitar nuestro estudio asumiremos que tan sólo disponemos del tipo entero.
- Asignación unaria: **x := op y**, donde **op** es una operación unaria. Las operaciones unarias principales incluyen el menos unario, la negación lógica, los operadores de desplazamiento de bits y los operadores de conversión de tipos.
- Asignación simple o copia: **x := y**, donde el valor de **y** se asigna a **x**.
- Salto incondicional: **goto etiqueta**.
- Saltos condicionales: **if x oprelacional y goto etiqueta**.

Como puede deducirse de los tercetos propuestos, las direcciones de memoria serán gestionadas de forma simbólica, esto es, a través de nombres en lugar de números. Por supuesto, para generar tercetos más cercanos a un código máquina general, cada uno de estos nombres debe ser traducido a una dirección de memoria única. Las direcciones de memoria se tomarían de forma consecutiva teniendo en cuenta el tamaño del tipo de datos que se supone alberga cada una de ellas; por

ejemplo, se puede asumir que un valor entero ocupa 16 bits, uno real ocupa 32 bits, un carácter ocupa 8 bits, etc.

Con los tercetos anteriores cubriremos todos los ejemplos propuestos en el presente capítulo. No obstante, estos tercetos no recogen todas las posibilidades de ejecución básicas contempladas por un microprocesador actual. Por ejemplo, para llamar a una subrutina se tienen tercetos para meter los parámetros en la pila de llamadas, para invocar a la subrutina indicando el número de parámetros que se le pasa, para tomar un parámetro de la pila, y para retornar:

- **param x**: mete al parámetro real **x** en la pila.
- **call p, n**: llama a la subrutina que comienza en la etiqueta **p**, y le dice que tome **n** parámetros de la cima de la pila.
- **pop x**: toma un parámetro de la pila y lo almacena en la dirección **x**.
- **return y**: retorna el valor **y**.

Otros tercetos permiten gestionar el direccionamiento indirecto y el indexado:

- Direccionamiento indexado: los tercetos son de la forma **y := x[i]** y **x[i] := y**, donde **x** es la dirección base, e **i** es el desplazamiento.
- Direccionamiento indirecto: los tercetos son de la forma **y:=&x** y **x:=*y**, donde el símbolo **&** quiere decir "la dirección de ..." y el símbolo ***** quiere decir "el valor de la dirección contenida en la dirección ...".

La elección de operadores permisibles es un aspecto importante en el diseño de código intermedio. El conjunto de operadores debe ser lo bastante rico como para permitir implementar todas las operaciones del lenguaje fuente. Un conjunto de operadores pequeño es más fácil de implantar en una nueva máquina objeto, pero si es demasiado limitado puede obligar a la etapa inicial a generar largas secuencias de instrucciones para algunas operaciones del lenguaje fuente. En tal caso, el optimizador y el generador de código tendrán que trabajar más si se desea producir un buen código.

A continuación se muestra un ejemplo de código de tercetos que almacena en la variable **c** la suma de **a** y **b** (nótese que **b** se destruye como consecuencia de este cálculo):

```
        c = a
label etqBucle
        if b = 0 goto etqFin
        b = b-1
        c = c+1
        goto etqBucle
label etqFin
```

8.3 Una calculadora simple compilada

Para ilustrar como se utiliza el código de tercetos en una gramática vamos a suponer que nuestra calculadora en lugar de ser una calculadora interpretada es una calculadora compilada, es decir, en vez de interpretar las expresiones vamos a generar código intermedio equivalente.

8.3.1 Pasos de construcción

Antes de comenzar a codificar los analizadores léxico y sintáctico es necesario plantear exactamente qué se desea hacer y con qué gramática. Para ello se debe proponer un ejemplo preliminar de lo que debe hacer la calculadora para, a continuación, crear la gramática que reconozca el lenguaje, asignar los atributos necesarios y, una vez claro el cometido de las acciones semánticas, comenzar la codificación.

8.3.1.1 Propuesta de un ejemplo

El objetivo es que la calculadora produzca un texto de salida ante un texto de entrada. Por ejemplo, si la entrada es:

```
a := 5*b;
c := b := d := a*(4+v);
c := c := c;
```

la salida debería ser:

```
tmp1=5*b
a=tmp1
tmp2=4+v
tmp3=a*tmp2
d=tmp3
b=d
c=b
c=c
c=c
```

que es el código de tercetos equivalente.

8.3.1.2 Gramática de partida

La gramática de partida basada en reglas de producción es:

```
prog    :   asig ';'
        |   prog asig ';'
        |   error ';'
        |   prog error ';'
        ;
asig    :   ID ASIG expr
        |   ID ASIG asig
        ;
expr    :   expr '+' expr
        |   expr '*' expr
        |   '(' expr ')'
        |   ID
        |   NUMERO
        ;
```

Como puede observarse, no se permite el programa vacío, lo que hace que sea necesario introducir una nueva regla de error que gestione la posibilidad de que se produzca algún fallo en la primera asignación. Nótese que si se produce un error en esta

primera asignación, la pila del análisis sintáctico aún no tiene el no terminal **prog** a su izquierda, por lo que no es posible reducir por la regla **prog : prog error ';'**, pero sí por la regla **prog : error ';'**.

Por otro lado, la finalización de cada sentencia se hace con un punto y coma ';', en lugar de con retorno de carro, lo que resulta más cercano al tratamiento real de los lenguajes de programación actuales.

Además de permitirse las asignaciones múltiples, la última diferencia con respecto a la gramática del apartado 7.3.1 radica en que se ha hecho caso omiso de los tipos y, por tanto, tampoco tienen sentido las funciones de conversión de un tipo a otro.

Por último, esta gramática expresada en notación BNF es:

```
gramatica():{}{
        (sentFinalizada())*
}
sentFinalizada():{}{
        asigCompuesta() <PUNTOYCOMA>
    |   /* Gestión del error */
}
asigCompuesta():{}{
        LOOKAHEAD(4)
        <ID> ":=" asigCompuesta()
    |   <ID> ":=" expr()
}
expr():{}{
        term() ( "+" term() )*
}
term():{}{
        fact() ( "*" fact() )*
}
fact():{}{
        <ID>
    |   <NUMERO>
    |   "(" expr() ")"
}
```

8.3.1.3 Consideraciones importantes sobre el traductor

Para facilitar la lectura del código intermedio resultante, se hará uso de identificadores de variables en lugar de utilizar directamente sus direcciones de memoria. En cualquier caso, no hay que olvidar que la tabla de símbolos debe contener, junto con el identificador de cada variable, la dirección de memoria de cada una de ellas, con objeto de realizar la traducción correspondiente en el momento de generar el código máquina final.

La gramática que se va a utilizar será prácticamente la misma que la empleada en apartados anteriores. Recordemos que en cada regla en la que aparece un operador aritmético intervienen sólo dos expresiones, lo que encaja con la generación de un

terceto de asignación binaria, en el que sólo hay dos operandos y un resultado. Sin embargo, la reducción por cada una de estas reglas representa un resultado intermedio en el camino de calcular el resultado final, tal y como ilustra la figura 8.3.

Estos resultados intermedios deben almacenarse en algún lugar para que los tercetos puedan trabajar con ellos. Para este propósito se utilizan variables temporales generadas automáticamente por el compilador en la cantidad que sea necesario. La figura 8.2 muestra el código de tercetos que se debe generar para una asignación que tiene como r-valor la expresión de la figura 8.3, así como las variables temporales que es necesario generar en cada paso. Como resulta evidente, el código de tercetos resultante es equivalente al código fuente.

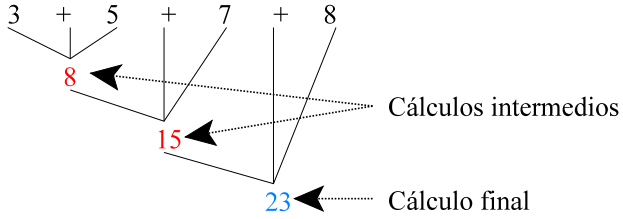


Figura 8.3. Los resultados intermedios se producen cada vez que se reduce por una regla de producción, e intervienen decisivamente en el cálculo del resultado final

Mediante el mecanismo de las variables temporales es posible construir, paso a paso, cualquier cálculo aritmético complejo, en el que pueden intervenir paréntesis,

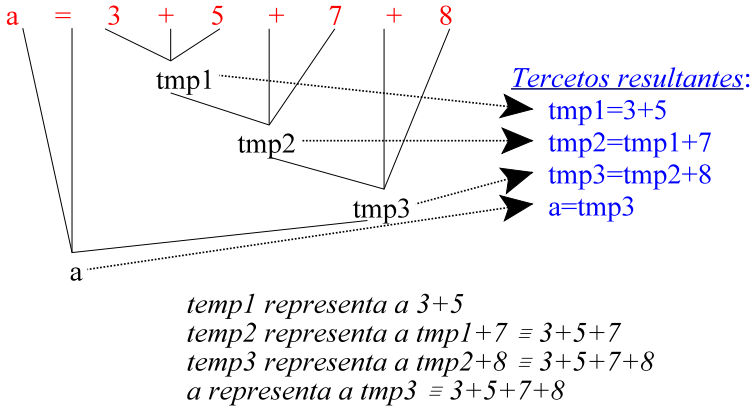


Figura 8.2. Representación de la variable temporal que hay que generar en cada reducción, y del terceto que debe producirse. El objetivo final de estos tercetos es almacenar en la variable **a** el valor **3+5+7+8**, lo que se consigue en base a las variables temporales

otras variables, etc. También resulta interesante observar que las variables temporales pueden reutilizarse en cálculos diferentes. No obstante, esto supone una optimización que no implementaremos en aras de concentrarnos exclusivamente en la generación de código correcto.

Para la realización de nuestra calculadora, hay que observar dos aspectos muy importantes:

- Gestionar adecuadamente los atributos. En la calculadora no hay parte de declaraciones ni chequeo de tipos, ya que se supone que todas las variables son de tipo numérico y que son declaradas en el momento en que hacen su primera aparición. Es por ello que no será necesaria una tabla de símbolos ya que, además, en los tercetos aparecerán los identificadores de las variables, y no las direcciones de memoria a que representan. Esto también nos permite omitir un gestor de direcciones de memoria que las vaya asignando a medida que van apareciendo nuevas variables.
- Realizar una secuencia de **printf**. Nuestro propósito es hacer una traducción textual, en la que entra un texto que representa asignaciones de alto nivel, y sale otro texto que representa a los tercetos. Por tanto nuestras acciones semánticas tendrán por objetivo realizar una secuencia de printf o System.out.println para visualizar estos textos de salida. Por este motivo, en tiempo de compilación, no se van a evaluar las expresiones ni a realizar cálculos de ningún tipo. Nuestro traductor se encarga únicamente de gestionar cadenas de texto; esto conlleva que, por ejemplo, el atributo asociado al token NUM no tenga porqué ser de tipo int, sino que basta con que sea un char[] o un String, ya que no va a ser manipulado aritméticamente.

En la construcción de un compilador real, la salida no suele ser un texto, sino un fichero binario que contiene el código máquina agrupado por bloques que serán gestionados por un enlazador (*linker*) para obtener el fichero ejecutable final. Asimismo, las acciones semánticas deben realizar toda la gestión de tipos necesaria antes de proceder a la generación de código, lo que lleva a la necesidad de una tabla de símbolos completa.

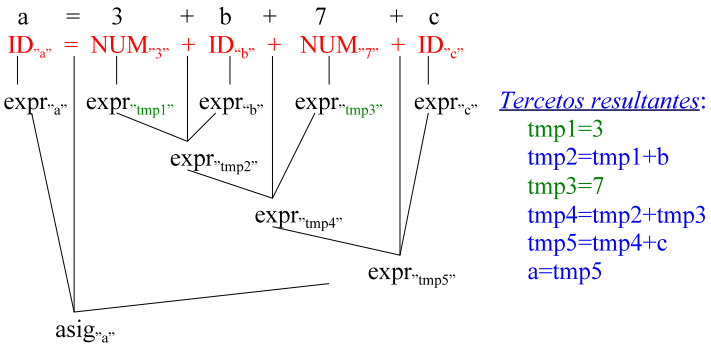
8.3.1.4 Atributos necesarios

Como ya se ha comentado, toda la gestión del código de entrada va a ser simbólica, en el sentido de que sólo se va a trabajar con cadenas de texto. De esta forma, tanto el terminal **ID** como el terminal **NUM** tendrán como atributo una cadena que almacena el lexema a que representan.

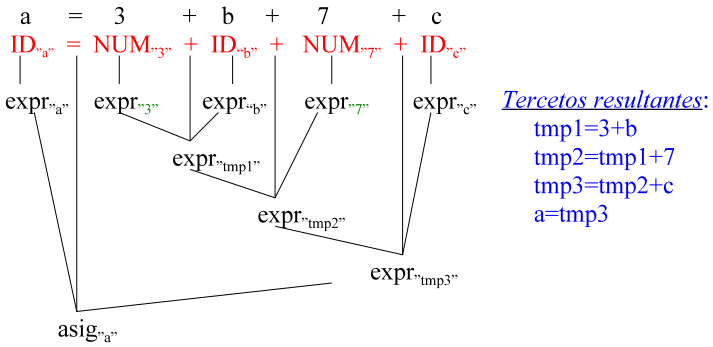
En cuanto a las expresiones, resulta sensato pensar que también deben tener como atributo a una cadena de caracteres que, de alguna forma, represente el trozo de árbol sintáctico del cual es raíz esa expresión. Retomando, por ejemplo, el árbol de la figura 8.2, se puede pensar en asignar a una expresión el texto que representa a cada variable, ya sea temporal o no; si asociamos el mismo tipo de atributo al no terminal

asig, entonces también se podrá generar código de tercetos incluso para las asignaciones múltiples. En cuanto a las expresiones que representan a un único valor numérico, se puede optar por generar una variable temporal para almacenarlos, o bien guardar el lexema numérico como atributo de la expresión, lo que conduce a una optimización en cuanto al número de tercetos generados y de variables temporales utilizadas. La figura 8.4.a) muestra el caso estricto en que sólo se permite como atributo el nombre de una variable, y la figura 8.4.b) muestra el caso relajado. En ambas situaciones el tipo de atributo es el mismo. En la solución que se propone más adelante se opta por el caso b), ya que supone una optimización sin ningún esfuerzo por parte del desarrollador.

Para acabar, la figura 8.5 muestra la propagación de atributos en el caso de realizar una asignación múltiple. Como ya se ha comentado antes, la generación de código se hace posible mediante la utilización de un atributo al no terminal **asig** que



Caso a). Una expresión representa siempre a una variable



Caso b). Una expresión representa a una variable o a un número

Figura 8.4. Asignación de atributos a los terminales y no terminales de la gramática de la calculadora. Em ambos casos el tipo de los atributos es el mismo. La diferencia entre el caso a) y el b) estriba en si se admite o no que una expresión pueda representar directamente a un literal numérico

representa al l-valor, en previsión de que sea, a su vez, el r-valor de una asignación múltiple que se reduce de derecha a izquierda.

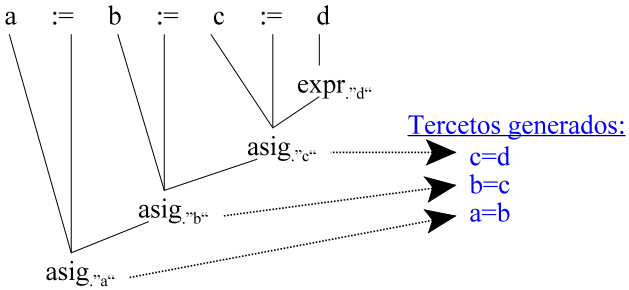


Figura 8.5. Asignaciones múltiples y el código generado equivalente

8.3.1.5 Acciones semánticas

Las acciones semánticas son sumamente sencillas, al no existir ninguna tabla de símbolos, ni haber chequeo de tipos. Tan sólo deben gestionarse convenientemente los atributos, propagándolos cuando sea necesario, generar las variables temporales y realizar los **printf** necesarios.

8.3.2 Solución con Lex/Yacc

En esta solución, el programa Lex se limita a reconocer los lexemas necesarios y a asignar a los *tokens* **NUM** e **ID** los lexemas que representan. El retorno de carro se considera un separador más. El código es:

```
%%
[0-9]+    {
                strcpy(yylval.cadena, yytext);
                return NUMERO;
          }

":="      { return ASIG; }
[a-zA-Z][a-zA-Z0-9]*   {
                                strcpy(yylval.cadena, yytext);
                                return ID;
                        }
[ \t\n]+  {;}
.         { return yytext[0]; }
```

El código Yacc también es muy escueto. Básicamente, cada reducción a una expresión o a una asignación, se encarga de generar un terceto, haciendo uso de variables temporales, si son necesarias. Éstas se crean mediante la función **nuevaTmp**.

Por otro lado, dado que todos los atributos son de tipo cadena de caracteres podría pensarse en hacer uso directo de **YYSTYPE** en lugar de **%union**. Sin embargo, la herramienta PCLex no permite asignar a YYSTYPE punteros estáticos (un array de

caracteres es, en C, un puntero estático), por lo que es necesario recurrir a la utilización del **%union** aunque ésta contenga un solo campo. El código resultante es:

```
%{
#include "stdio.h"
void nuevaTmp(char * s){
        static int actual=1;
        sprintf(s, "tmp%d", actual++);
}
%}
%union{
        char cadena[50];
}
%token <cadena> NUMERO ID
%token ASIG
%type <cadena> asig expr
%start prog
%left '+'
%left '*'
%%
prog    :   asig ';'
        |   prog asig ';'
        |   error ';'           { yyerrok; }
        |   prog error ';'      { yyerrok; }
        ;
asig    :   ID ASIG expr        {
                                  printf("%s=%s\n", $1, $3);
                                  strcpy($$, $1);
                                }
        |   ID ASIG asig        {
                                  printf("%s=%s\n", $1, $3);
                                  strcpy($$, $1);
                                }
        ;
expr    :   expr '+' expr {
                            nuevaTmp($$);
                            printf("%s=%s+%s\n", $$, $1, $3);
                          }
        |   expr '*' expr  {
                            nuevaTmp($$);
                            printf("%s=%s*%s\n", $$, $1, $3);
                          }
        |   '(' expr ')'    { strcpy($$, $2); }
        |   ID              { strcpy($$, $1); }
        |   NUMERO          { strcpy($$, $1); }
        ;
%%
#include "Calcul.c"
void main(){
        yyparse();
```

```
48 }
49 void yyerror(char * s){
50         printf("%s\n",s);
51 }
```

8.3.3 Solución con JFlex/Cup

La solución con JFlex y Cup es casi idéntica a la ya vista en Lex/Yacc. Quizás la única diferencia radica en la función **nuevaTmp**, que hace uso de la variable estática **actual** inicializada a 1 en lugar de a 0 para facilitar la concatenación con el literal "tmp".

Así pues, el código en JFlex es:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  %%
4  %unicode
5  %cup
6  %line
7  %column
8  %%
9  "+"        { return new Symbol(sym.MAS); }
10 "*"        { return new Symbol(sym.POR); }
11 "("        { return new Symbol(sym.LPAREN); }
12 ")"        { return new Symbol(sym.RPAREN); }
13 ";"        { return new Symbol(sym.PUNTOYCOMA); }
14 ":="       { return new Symbol(sym.ASIG); }
15 [:jletter:][:jletterdigit:]*        { return new Symbol(sym.ID, yytext()); }
16 [:digit:]+              { return new Symbol(sym.NUMERO, yytext()); }
17 [ \t\r\n]+                         {;}
18 .                       { System.out.println("Error léxico."+yytext()+"-"); }
```

Por su parte, el código Cup queda:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  parser code {:
4          public static void main(String[] arg){
5              Yylex miAnalizadorLexico =
6                      new Yylex(new InputStreamReader(System.in));
7              parser parserObj = new parser(miAnalizadorLexico);
8               try{
9                  parserObj.parse();
10             }catch(Exception x){
11                 x.printStackTrace();
12                 System.out.println("Error fatal.\n");
13             }
14         }
15 :};
16 action code {:
17         private static int actual=0;
```

```
18          private static String nuevaTmp(){
19              return "tmp"+(++actual);
20          }
21  :}
22  terminal PUNTOYCOMA, MAS, POR;
23  terminal ASIG, LPAREN, RPAREN;
24  terminal String ID, NUMERO;
25  non terminal String asig, expr;
26  non terminal prog;
27  precedence left MAS;
28  precedence left POR;
29  /* Gramática */
30  prog    ::=  asig PUNTOYCOMA
31          |    prog asig PUNTOYCOMA
32          |    error PUNTOYCOMA
33          |    prog error PUNTOYCOMA
34          ;
35  asig    ::=  ID:i ASIG expr:e  {:
36                                   System.out.println(i+"="+e);
37                                   RESULT=i;
38                                 :}
39          |    ID:i ASIG asig:a  {:
40                                   System.out.println(i+"="+a);
41                                   RESULT=i;
42                                 :}
43          ;
44  expr    ::=  expr:e1 MAS expr:e2 {:
45                                     RESULT=nuevaTmp();
46                                     System.out.println(RESULT+"="+e1+"+"+e2);
47                                   :}
48          |    expr:e1 POR expr:e2 {:
49                                     RESULT=nuevaTmp();
50                                     System.out.println(RESULT+"="+e1+"*"+e2);
51                                   :}
52          |    LPAREN expr:e RPAREN {: RESULT=e; :}
53          |    ID:i                 {: RESULT=i; :}
54          |    NUMERO:n             {: RESULT=n; :}
55          ;
```

8.3.4 Solución con JavaCC

La solución con JavaCC sigue exactamente las mismas pautas que las soluciones ascendentes. Quizás la única diferencia radica en que el código está más estructurado, al haberse ubicado en funciones independientes cada una de las acciones semánticas lo que, entre otras cosas, permite dar un tratamiento uniforme a las operaciones aritméticas mediante la función **usarOpAritmetico()**. Por otro lado, el reconocimiento de asignaciones múltiples se ha hecho mediante una regla recursiva por la derecha y utilizando el LOOKAHEAD necesario (4 en este caso).

El código completo es:

```
1  PARSER_BEGIN(Calculadora)
2        import java.util.*;
3        public class Calculadora{
4              private static int actual=0;
5              public static void main(String args[]) throws ParseException {
6                    new Calculadora(System.in).gramatica();
7              }
8              private static void usarASIG(String s, String e){
9                    System.out.println(s+"="+e);
10             }
11             private static String usarOpAritmetico(String e1, String e2, String op){
12                   actual++;
13                   String tmp="tmp"+actual;
14                   System.out.println(tmp+"="+e1+op+e2);
15                   return tmp;
16             }
17       }
18 PARSER_END(Calculadora)
19 SKIP : {
20             " "
21       |     "\t"
22       |     "\r"
23       |     "\n"
24 }
25 TOKEN [IGNORE_CASE] :
26 {
27             <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
28       |     <NUMERO: (["0"-"9"])+>
29       |     <PUNTOYCOMA: ";">
30 }
31 /*
32 gramatica ::= ( sentFinalizada )*
33 */
34 void gramatica():{}{
35       (sentFinalizada())*
36 }
37 /*
38 sentFinalizada ::= ( ID ASIG )+ expr ';' | error ';'
39 */
40 void sentFinalizada():{}{
41       try {
42             asigCompuesta() <PUNTOYCOMA>
43       }catch(ParseException x){
44             System.out.println(x.toString());
45             Token t;
46             do {
47                   t = getNextToken();
48             } while (t.kind != PUNTOYCOMA);
```

```
49      }
50  }
51  String asigCompuesta():{
52      String s;
53      String a, e;
54  }{  LOOKAHEAD(4)
55          s=id() ":=" a=asigCompuesta()  { usarASIG(s, a); return s; }
56      |   s=id() ":=" e=expr()           { usarASIG(s, e); return s; }
57  }
58  /*
59  expr ::= term ('+' term)*
60  */
61  String expr():{
62      String t1, t2;
63  }{
64      t1=term() ( "+" t2=term() { t1=usarOpAritmetico(t1, t2, "+"); } )*      { return t1; }
65  }
66  /*
67  term ::= fact ('*' fact)*
68  */
69  String term():{
70      String f1, f2;
71  }{
72      f1=fact() ( "*" f2=fact() { f1=usarOpAritmetico(f1, f2, "*"); } )*      { return f1; }
73  }
74  /*
75  fact ::= ID | NUMERO | '(' expr ')'
76  */
77  String fact():{
78      String s, e;
79  }{
80          s=id()              { return s; }
81      |   s=numero()          { return s; }
82      |   "(" e=expr() ")"    { return e; }
83  }
84  String id():{}{
85      <ID>            { return token.image; }
86  }
87  String numero():{}{
88      <NUMERO> { return token.image; }
89  }
90  SKIP : {
91      <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
92  }
```

Aunque los no terminales **sentFinalizada()** y **asigCompuesta()** podrían haberse fusionado en uno sólo, se ha preferido diferenciar claramente entre la gestión del error a través del terminal **<PUNTOYCOMA>**, y el reconocimiento de una asignación compuesta con recursión a derecha. Nótese asimismo que el no terminal

asigCompuesta() engloba también el concepto de asignación simple como caso base de su recursión

8.4 Generación de código en sentencias de control

Una vez estudiado el mecanismo general de generación de código en expresiones aritméticas, el siguiente paso consiste en abordar el estudio del código de tercetos equivalente a las sentencias de alto nivel para el control del flujo de ejecución. Entre este tipo de sentencias están los bucles WHILE y REPEAT, así como las sentencias condicionales IF y CASE.

Como vimos en el apartado 8.2, el control del flujo de ejecución se realiza a bajo nivel mediante los tercetos **goto** y **call**. En este texto no entraremos en detalle sobre la gestión de subrutinas, por lo que nos centraremos exclusivamente en el **goto** incondicional y condicional.

8.4.1 Gramática de partida

Nuestro propósito consiste en reconocer programas que permitan realizar asignaciones a variables de tipo entero, y realizar cambios de flujo en base a la evaluación de condiciones. Las condiciones pueden ser simples o complejas; en las simples sólo aparecen operadores relacionales: >, <, >=, etc.; en las complejas se pueden utilizar los operadores lógicos **AND**, **OR** y **NOT** con el objetivo de concatenar condiciones más básicas.

Como puede observarse, se permiten bucles de tipo **WHILE** y de tipo **REPEAT**. En los primeros se evalúa una condición **antes** de entrar al cuerpo del bucle; si ésta es cierta se ejecuta el cuerpo y se comienza de nuevo el proceso; en caso contrario se continúa por la siguiente sentencia tras el **WHILE**. Por tanto el cuerpo se ejecuta **mientras** la condición que acompaña al **WHILE** sea **cierta**. Por otro lado, el bucle **REPEAT** evalúa la condición **tras** la ejecución del cuerpo, de forma que éste se ejecuta **hasta** que la condición del bucle sea **cierta**.

Como sentencias condicionales se permite el uso de la cláusula **IF** que permite ejecutar un bloque de código de forma opcional, siempre y cuando la condición sea **cierta**. Asimismo, un **IF** puede ir acompañado de una parte **ELSE** opcional que se ejecutaría caso de que la condición fuese falsa. Por otro lado, también se ha incluido la sentencia **CASE** que evalúa una expresión aritmética y la compara con cada uno de los elementos de una lista de valores predeterminada; caso de coincidir con alguno de ellos pasa a ejecutar el bloque de código asociado a dicho valor. Si no coincide con ninguno de ellos, pasa a ejecutar (opcionalmente) el bloque de código indicado por la cláusula **OTHERWISE**. Por último, se ha decidido no incluir la sentencia **FOR** con objeto de que el lector pruebe a codificar su propia solución, ya que resulta un ejercicio interesante.

Para ilustrar los mecanismos puros de generación de código, y no entremezclar en las acciones semánticas ninguna acción relativa a la gestión de tablas de símbolos

ni control de tipos, se ha decidido que nuestro pequeño lenguaje sólo posea el tipo entero y que no sea necesario declarar las variables antes de su utilización.

La gramática que se va a utilizar, en formato de reglas de producción, es:

```
prog        :       prog sent ';'
            |       prog error ';'
            |       /* Épsilon */
            ;
sent        :       ID ASIG expr
            |       IF cond THEN sent ';' opcional FIN IF
            |       '{' lista_sent '}'
            |       WHILE cond DO sent ';' FIN WHILE
            |       REPEAT sent ';' UNTIL cond
            |       sent_case
            ;
opcional            :       ELSE sent';'
                    |       /*Epsilon*/
                    ;
lista_sent          :       /*Epsilon*/
                    |       lista_sent sent ';'
                    |       lista_sent error ';'
                    ;
sent_case           :       inicio_case OTHERWISE sent ';' FIN CASE
                    |       inicio_case FIN CASE
                    ;
inicio_case         :       CASE expr OF
                    |       inicio_case CASO expr ':'   sent ';'
                    ;
expr        :       NUMERO
            |       ID
            |       expr'+'expr
            |       expr'-'expr
            |       expr'*'expr
            |       expr'/'expr
            |       '-'expr %prec MENOS_UNARIO
            |       '('expr')'
            ;
cond        :       expr'>'expr
            |       expr'<'expr
            |       expr'MAI'expr
            |       expr'MEI'expr
            |       expr'='expr
            |       expr'DIF'expr
            |       NOT cond
            |       cond AND cond
            |       cond OR cond
            |       '(' cond ')'
            ;
```

Es de notar que el cuerpo de una sentencia **WHILE**, **REPEAT**, **IF**, etc. está formado por una sola sentencia. Si se desea incluir más de una, el mecanismo consiste en recurrir a la sentencia compuesta. Una sentencia compuesta está formada por una lista de sentencias delimitada por llaves, tal y como indica la regla:

```
sent :    '{' lista_sent '}'
```

Además, nada impide introducir como cuerpo de cualquier sentencia de control de flujo a otra sentencia de control de flujo. Es más, el anidamiento de sentencias puede producirse a cualquier nivel de profundidad. Este hecho hace que haya que prestar especial atención al control de errores. Como puede observarse en la gramática, se tienen dos reglas de error, una a nivel de programa y otra a nivel de **lista_sent**, esto es, en el cuerpo de una sentencia compuesta.

Si se produce un error sintáctico en el interior de una sentencia compuesta, y sólo se tuviera la regla de error a nivel de programa, se haría una recuperación incorrecta, ya que el mecanismo *panic mode* extraería elementos de la pila hasta encontrar **prog**; luego extraería *tokens* hasta encontrar el punto y coma. Y por último continuaría el proceso de análisis sintáctico pensando que ha recuperado el error convenientemente y que las siguientes sentencias están a nivel de programa, lo cual es falso puesto que se estaba en el interior de una sentencia compuesta. Este problema se ilustra en la figura 8.6. Para evitarlo es necesario introducir una nueva regla de error a nivel de sentencia compuesta.

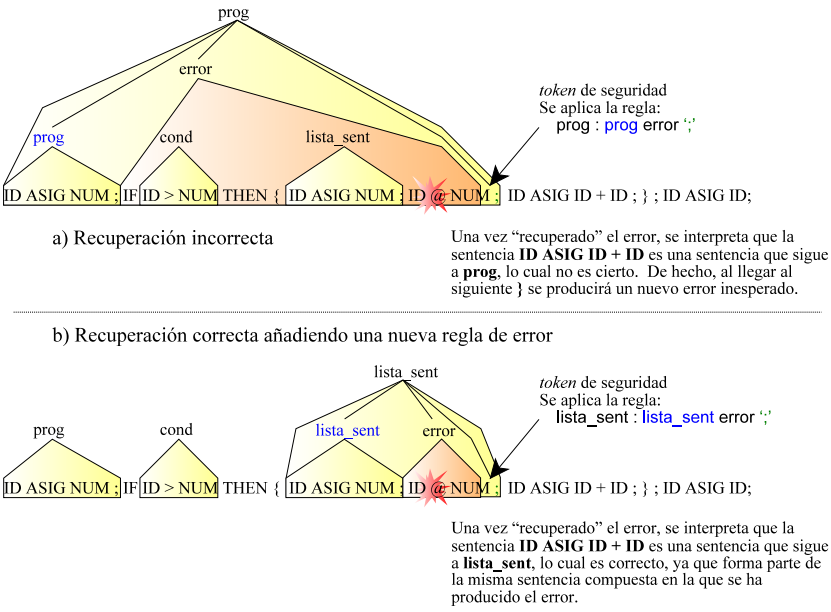


Figura 8.6. Necesidad de dos reglas de error cuando se utilizan sentencias compuestas

Desde el punto de vista BNF para JavaCC, la gramática queda:

```
void gramatica():{}{
        (sentFinalizada())*
}
void sentFinalizada():{}{
        (
                s=id() <ASIG> e=expr()
        |       <IF> cond() <THEN> sentFinalizada()
                [<ELSE> sentFinalizada()]
                <FIN> <IF>
        |       <WHILE> cond() <DO>
                        sentFinalizada()
                <FIN> <WHILE>
        |       <REPEAT>
                        sentFinalizada()
                <UNTIL> cond()
        |       <LLLAVE> gramatica() <RLLAVE>
        |       <CASE> expr() <OF>
                        (<CASO> expr() <DOSPUNTOS> sentFinalizada() )*
                        [<OTHERWISE> sentFinalizada()]
                <FIN> <CASE>
        ) <PUNTOYCOMA>
        | /* Gestión del error */
}
String expr():{}{
        term()    ( <MAS  | <MENOS> term() )*
}
String term():{}{
        fact()    ( <POR> | <ENTRE> fact() )*
}
String fact():{}{
        (<MENOS>)*
        (
                <ID>
        |       <NUMERO>
        |       <LPAREN> expr() <RPAREN>
        )
}
BloqueCondicion cond():{}{
        condTerm()        ( <OR> condTerm() )*
}
BloqueCondicion condTerm():{}{
        c1=condFact()     ( <AND> condFact() )*
}
BloqueCondicion condFact():{}{
        (<NOT> )*
        (       condSimple()
        |       <LCOR> cond() <RCOR>
        )
```

```
}
BloqueCondicion condSimple():{}{
        expr()    (                   <MAYOR> expr()
                                  |   <MENOR> expr()
                                  |   <IGUAL> expr()
                                  |   <MAI> expr()
                                  |   <MEI> expr()
                                  |   <DIF> expr()
                                  )
}
```

Esta gramática tiene dos diferencias con respecto a la expresada con reglas de producción. En primer lugar el no terminal **lista_sent** ha desaparecido, ya que sintácticamente es equivalente a **prog** (que aquí hemos llamado **gramática**). Ello nos lleva a necesitar un solo control de errores en la última regla del no terminal **sentFinalizada**. Evidentemente, esto también podría haberse hecho en la gramática basada en reglas, pero se ha preferido ilustrar la gestión de varias reglas de error en una misma gramática.

La segunda diferencia radica en la gestión de las condiciones mediante una gramática descendente. Ante una secuencia de caracteres como: "((((27*8) ..." es necesario conocer el contenido de los puntos suspensivos para poder decidir si los tres primeros paréntesis se asocian a una expresión o a una condición: si nos encontramos el carácter ">" los paréntesis se asocian a una condición, pero si nos encontramos "-7)" entonces, al menos, el tercer paréntesis es de una expresión y aún habría que seguir mirando para ver a qué pertenecen los otros dos paréntesis. Este problema tiene tres soluciones claramente diferenciadas:

- Establecer un *lookahead* infinito.
- Diferenciar las agrupaciones de expresiones y las de condiciones; por ejemplo, las expresiones se pueden agrupar entre paréntesis, y las condiciones entre corchetes. Esta ha sido la solución adoptada en este ejemplo.
- Considerar que una condición es una expresión de tipo lógico o *booleano*. Esta es la decisión más ampliamente extendida, aunque no se ha elegido en este caso ya que implica una gestión de tipos sobre la que no nos queremos centrar.

8.4.2 Ejemplos preliminares

A continuación se ilustran unos cuantos ejemplos de programas válidos reconocidos por nuestra gramática, así como el código de tercetos equivalente que sería deseable generar.

El primer ejemplo ilustra el cálculo iterativo del factorial de un número **n**:

```
ALFA := n;
FACTORIAL := 1;
WHILE ALFA > 1 DO
```

```
        {
         FACTORIAL := FACTORIAL * ALFA;
         ALFA := ALFA - 1;
        };
        FIN WHILE;
```

El código de tercetos equivalente sería:

```
ALFA = n
        tmp1 = 1;
        FACTORIAL = tmp1
label etq1
        tmp2 = 1;
        if ALFA > tmp2 goto etq2
        goto etq3
label etq2
        tmp3 = FACTORIAL * ALFA;
        FACTORIAL = tmp3
        tmp4 = 1;
        tmp5 = ALFA - tmp4;
        ALFA = tmp5
        goto etq1
label etq3
```

El siguiente ejemplo ilustra el código generado para una sentencia CASE:

```
CASE NOTA OF
 * Podemos emplear comentarios, dedicando una linea a cada uno de ellos.
 CASO 5 : CALIFICACION := SOBRESALIENTE;
 CASO 4 : CALIFICACION := NOTABLE;
 CASO 3 : CALIFICACION := APROBADO;
 CASO 2 : CALIFICACION := INSUFICIENTE;
OTHERWISE
 CALIFICACION := MUY_DEFICIENTE;
FIN CASE;
```

que equivale a:

```
tmp1 = 5;
        if NOTA != tmp1 goto etq2
        CALIFICACION = SOBRESALIENTE
        goto etq1
label etq2
        tmp2 = 4;
        if NOTA != tmp2 goto etq3
        CALIFICACION = NOTABLE
        goto etq1
label etq3
        tmp3 = 3;
        if NOTA != tmp3 goto etq4
        CALIFICACION = APROBADO
        goto etq1
label etq4
        tmp4 = 2;
```

```
        if NOTA != tmp4 goto etq5
        CALIFICACION = INSUFICIENTE
        goto etq1
label etq5
        CALIFICACION = MUY_DEFICIENTE
label etq1
```

Para finalizar, el siguiente ejemplo demuestra cómo es posible anidar sentencias compuestas a cualquier nivel de profundidad:

```
JUGAR := DESEO_DEL_USUARIO;
WHILE JUGAR = VERDAD DO
 {
  TOTAL := 64;
  SUMA_PUNTOS := 0;
  NUMERO_TIRADAS := 0;
  TIRADA_ANTERIOR := 0;
  REPEAT
   {
    DADO := RANDOMIZE * 5 + 1;
    IF TIRADA_ANTERIOR != 6 THEN
      NUMERO_TIRADAS := NUMERO_TIRADAS + 1;
    FIN IF;
    SUMA_PUNTOS := SUMA_PUNTOS + DADOS;
    IF SUMA_PUNTOS > TOTAL THEN
      SUMA_PUNTOS := TOTAL -(SUMA_PUNTOS - TOTAL);
    ELSE
      IF SUMA_PUNTOS != TOTAL THEN
        CASE DADO OF
          CASO 1: UNOS := UNOS + 1;
          CASO 2: DOSES := DOSES + 1;
          CASO 3: TRESES := TRESES + 1;
          CASO 4: CUATROS := CUATROS + 1;
          CASO 5: CINCOS := CINCOS + 1;
        OTHERWISE
              SEISES := SEISES + 1;
        FIN CASE;
      FIN IF;
    FIN IF;
    TIRADA_ANTERIOR := DADO;
   };
   UNTIL SUMA_PUNTOS = TOTAL;
   JUGAR := DESEO_DEL_USUARIO;
 };
FIN WHILE;
```

En este caso, el código generado es verdaderamente enrevesado:

```
      JUGAR = DESEO_DEL_USUARIO          label etq2
label etq1                                     tmp1 = 64;
      if JUGAR = VERDAD goto etq2              TOTAL = tmp1
      goto etq3                                tmp2 = 0;
```

```
        SUMA_PUNTOS = tmp2
        tmp3 = 0;
        NUMERO_TIRADAS = tmp3
        tmp4 = 0;
        TIRADA_ANTERIOR = tmp4
label etq4
        tmp5 = 5;
        tmp6 = RANDOMIZE * tmp5;
        tmp7 = 1;
        tmp8 = tmp6 + tmp7;
        DADO = tmp8
        tmp9 = 6;
        if TIRADA_ANTERIOR != tmp9 goto etq5
        goto etq6
label etq5
        tmp10 = 1;
        tmp11 = NUMERO_TIRADAS + tmp10;
        NUMERO_TIRADAS = tmp11
        gotoetq7
label etq6
label etq7
        tmp12 = SUMA_PUNTOS + DADOS;
        SUMA_PUNTOS = tmp12
        if SUMA_PUNTOS > TOTAL goto etq8
        goto etq9
label etq8
        tmp13 = SUMA_PUNTOS - TOTAL;
        tmp14 = TOTAL - tmp13;
        SUMA_PUNTOS = tmp14
        gotoetq10
label etq9
        if SUMA_PUNTOS != TOTAL goto etq11
        goto etq12
label etq11
        tmp15 = 1;
        if DADO != tmp15 goto etq14
        tmp16 = 1;
        tmp17 = UNOS + tmp16;
        UNOS = tmp17
        goto etq13
label etq14
        tmp18 = 2;
        if DADO != tmp18 goto etq15
        tmp19 = 1;
        tmp20 = DOSES + tmp19;
        DOSES = tmp20
        goto etq13
label etq15
        tmp21 = 3;
        if DADO != tmp21 goto etq16
        tmp22 = 1;
        tmp23 = TRESES + tmp22;
        TRESES = tmp23
        goto etq13
label etq16
        tmp24 = 4;
        if DADO != tmp24 goto etq17
        tmp25 = 1;
        tmp26 = CUATROS + tmp25;
        CUATROS = tmp26
        goto etq13
label etq17
        tmp27 = 5;
        if DADO != tmp27 goto etq18
        tmp28 = 1;
        tmp29 = CINCOS + tmp28;
        CINCOS = tmp29
        goto etq13
label etq18
        tmp30 = 1;
        tmp31 = SEISES + tmp30;
        SEISES = tmp31
label etq13
        goto etq19
label etq12
label etq19
label etq10
        TIRADA_ANTERIOR = DADO
        if SUMA_PUNTOS = TOTAL goto etq20
        goto etq21
label etq21
        goto etq4
label etq20
        JUGAR = DESEO_DEL_USUARIO
        goto etq1
label etq3
```

8.4.3 Gestión de condiciones

Todos los cambios de flujo que admite el lenguaje que acabamos de definir son condicionales. En otras palabras, en base a la veracidad o falsedad de una condición, el programa sigue su flujo de ejecución por una sentencia o por otra. Es por esto que el código que decidamos generar para evaluar una condición será decisivo para gestionar el flujo en las diferentes sentencias de control.

A este respecto, hay dos posibilidades fundamentales. Quizás la más sencilla estriba en considerar que las condiciones son expresiones especiales de tipo lógico y

cuyo valor de evaluación es 0 ó 1 (falso ó verdadero respectivamente) y se almacena en una variable temporal al igual que ocurre con las expresiones de tipo entero. La sentencia en la que se enmarca esta condición preguntará por el valor de la variable temporal para decidir por dónde continúa el flujo. Por ejemplo, la regla:

```
sent : IF cond THEN sent opcional FIN IF
                    ↑              ↑      ↑
                    ①              ②      ③
```

establecería en una acción ubicada en el punto ① un terceto del tipo

```
if tmpXX=0 goto etqFalso
```

donde **tmpXX** es la variable en la que se almacena el resultado de haber evaluado la condición. Si su valor es diferente de 0, o sea cuando la condición es verdad, entonces no se produce el salto y se continúa el flujo de ejecución, que se corresponde con la sentencia del cuerpo del IF. El resultado es que esta sentencia sólo se ejecuta cuando la condición es cierta. Como opcional se corresponde con el cuerpo del **ELSE** (posiblemente vacío), pues es aquí exactamente adonde se debe saltar en caso de que la condición sea falsa. En cualquier caso, una vez ejecutado el cuerpo del **IF** se debe saltar detrás del **FIN IF**, con objeto de no ejecutar en secuencia la parte opcional del **ELSE**. Por tanto, en el punto ② deben colocarse los tercetos:

```
        goto etqFINIF
label etqFalso:
```

Finalmente la etiqueta de fin del **IF** debe ubicarse detrás de la sentencia completa, en el punto ③:

```
label etqFINIF:
```

con lo que el código completo generado sería:

```
        // Código que evalúa cond y mete su valor en tmpXX
        if tmpXX=0 goto etqFalso
        // Código de la sentencia del IF
        goto etgFINIF
label etqFalso:
        // Código de la sentencia del ELSE
label etqFINIF:
```

Para realizar este propósito, resulta fundamental la inserción de acciones semánticas intercaladas en el consecuente de una acción semántica. Además, las etiquetas que aquí se han identificado como etqFalso y etqFINIF deben ser, en realidad, etiquetas cualesquiera generadas ad hoc por nuestro compilador, por ejemplo etq125 y etq563 respectivamente.

En cualquier caso, este mecanismo de evaluación de condiciones incurre en una disminución de la eficiencia ya que las condiciones compuestas deben evaluarse al completo. Para evitar este problema sin enrarecer el código, se utiliza una técnica que permite la evaluación de condiciones usando cortocircuito. Por ejemplo, en una condición compuesta de la forma **cond$_1$ OR cond$_2$**, sólo se evaluará la segunda condición si la primera es falsa ya que, de lo contrario, se puede asegurar que el resultado del **OR** será verdadero independientemente del valor de cond$_2$. Una cosa parecida sucede con el **AND**. Éste será el método que emplearemos, y que se explica

con detalle a continuación.

8.4.3.1 Evaluación de condiciones usando cortocircuito

El mecanismo consiste en que cada **cond** genere un bloque de código que posee dos saltos (**goto**): un salto a una etiqueta **A** caso de cumplirse la condición, y un salto a otra etiqueta **B** caso de no cumplirse la condición.

Generar este código para las condiciones simples resulta trivial. A partir de ellas se estudiará el cortocircuito en condiciones compuestas.

8.4.3.1.1 Condiciones simples

Como punto de partida se asume que las expresiones y las asignaciones generan el mismo código intermedio que se estudió en la calculadora del apartado 8.3. Por ello, ante una condición como por ejemplo:

7*a > 2*b-1

se utilizará la regla:

cond : $expr_1$ '>' $expr_2$

y por las reglas de **expr**, y sin introducir todavía ninguna acción semántica en la regla de **cond**, se genera el código:

❶ tmp1=7*a
❷ tmp2=2*b
❸ tmp3=tmp2-1

donde la línea ❶ se corresponde con $expr_1$ y las líneas ❷ y ❸ se corresponden con $expr_2$. Además, el atributo que representa a $expr_1$ tiene el valor "tmp1", mientras que quien representa a $expr_2$ es "tmp3".

Por otro lado, pretendemos que una regla de producción de una condición simple como:

cond : $expr_1$ op_{rel} $expr_2$

genere un bloque de código de la forma:

if var_1 op_{rel} var_2 goto etq_{Verdad}
goto etq_{Falso}

donde var_1 es el representante de la primera expresión, ya sea una variable temporal, una variable de usuario o una constante numérica; var_2 representa a la segunda expresión; y op_{rel} representa al operador de comparación: mayor que (>), menor que (<), distinto de (<>), etc. Además, etq_{Verdad} y etq_{Falso} son etiquetas generadas *ad hoc* por el compilador de manera secuencial: etq001, etq002, etc., de forma parecida a como se generaban las variables temporales. Siguiendo con nuestro ejemplo, el código que nos interesa producir es:

if tmp1 > tmp3 goto etq001
goto etq002

donde etq001 representa la etq_{Verdad} y etq002 representa etq_{Falso}. Para generar este código basta con asociar una acción semántica a la regla de la condición, de forma que ésta quedaría:

cond : $expr_1$ '>' $expr_2$ {

```
                nuevaEtq($$.etqVerdad);
                nuevaEtq($$.etqFalso);
                printf("\tif %s > %s goto %s\n", $1, $3, $$.etqVerdad);
                printf("\tgoto %s\n", $$.etqFalso);
        }
```

Como puede observarse en esta acción semántica, los atributos asociados a una condición son un par de etiquetas, una de verdad y otra de falso. Además, el bloque de código g enerado posee dos **gotos** (uno condicional y otro incondicional), a sendas etiquetas, pero no ha ubicado el destino de ninguna de ellas, esto es, no se han puesto los **label**. La figura 8.7 muestra el código asociado a cada bloque gramatical.

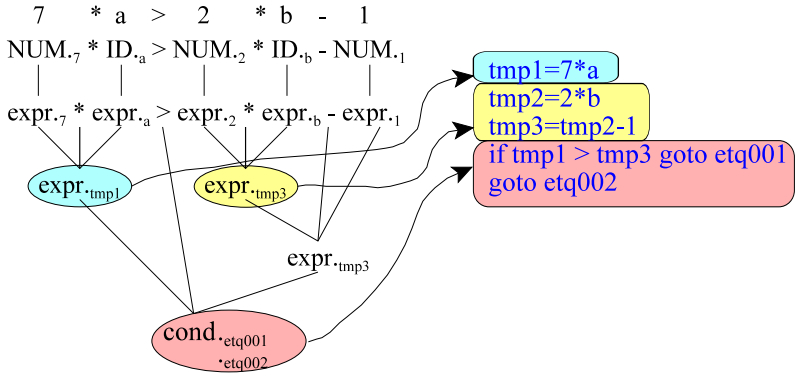


Figura 8.7. Generación de código para una condición simple

Esto se debe a que la regla que haga uso de la condición (ya sea en un IF, en un WHILE, etc.), debe colocar estos **label** convenientemente. Por ejemplo, un WHILE colocaría el destino de la etiqueta de verdad al comienzo del cuerpo del bucle, mientras que la de falso iría tras el cuerpo del bucle; de esta manera cuando la condición sea

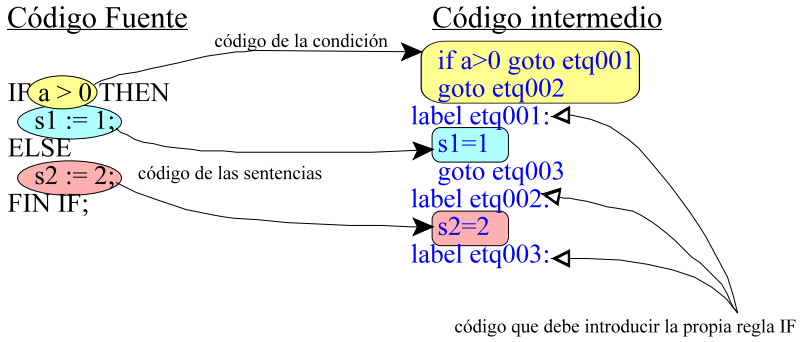


Figura 8.8. Código que debe generar una sentencia IF completa

falsa se salta el cuerpo, y cuando sea cierta se ejecuta un nuevo ciclo. Una cosa parecida ocurriría con el resto de las sentencias; la figura 8.8 muestra el caso de un **IF**.

8.4.3.1.2 Condiciones compuestas

Un hecho que se deduce de la discusión anterior es que las sentencias en las que se enmarca una condición (IF, WHILE, etc.) deben realizar un tratamiento uniforme de éstas tanto si se trata de condiciones simples como de condiciones compuestas en las que intervienen los operadores lógicos AND, OR y NOT. Por tanto, una condición compuesta debe generar por sí misma un bloque de código en el que existan dos **gotos** para los cuales no se ha establecido aún el destino. Las etiquetas de destino se asignarán como atributos del no terminal **cond**, con lo que el tratamiento de las condiciones compuestas coincide con el de las simples.

Además, se desea utilizar la técnica del cortocircuito, por el que una condición sólo se evalúa si su valor de verdad o falsedad es decisivo en la evaluación de la condición compuesta en la que interviene.

La explicación siguiente puede comprenderse mejor si pensamos en el bloque de código generado por una condición como una caja negra que ejecuta, en algún momento, o un salto a una etiqueta de falso, o uno a una etiqueta de verdad, tal y como ilustra la figura 8.9.a). Por tanto, ante una regla como

cond : $cond_1$ AND $cond_2$

se generarían los bloques de código de la figura 8.9.b). Pero, al tener que reducir todo el conjunto a una condición compuesta, los bloques de la figura 8.9.b) deben reducirse a uno sólo, en el que exista una sola etiqueta de verdad y una sola de falso, tal y como ilustra la figura 8.9.c). Asumimos que el flujo le llega a un bloque por arriba.

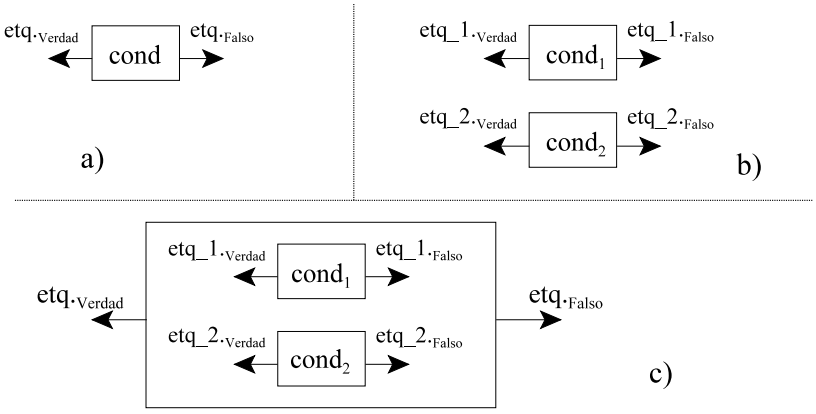


Figura 8.9. Diagramas de bloques asociados a las condiciones

En otras palabras, la figura 8.9.c) muestra que, sin asociar todavía ninguna

acción semántica a la regla

cond : $cond_1$ AND $cond_2$

se generarían bloques de código que poseen cuatros **gotos** pero ningún **label**. Por tanto nuestro objetivo es:

- Reducir los cuatro **gotos** sin **label** a tan sólo dos **gotos** y que coincidan con las etiquetas de verdad y falso del bloque cond general.
- Que la segunda condición sólo se evalúe cuando la primera es cierta; en caso contrario la técnica del cortocircuito no necesita evaluarla ya que la condición general es falsa.

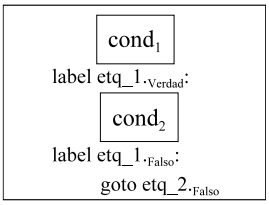


Figura 8.10. Código asociado a una condición compuesta por AND

De esta forma, el resultado general es que habremos generado un solo bloque de código grande con una sola etiqueta de verdad y una sola de falso. Al obtener un único gran bloque de código, desde el punto de vista externo podremos tratar una condición de manera uniforme, ya se trate de una condición simple o una compuesta.

Podemos conseguir estos dos objetivos de una manera muy sencilla; el método consiste en colocar los **label** de $cond_1$ estratégicamente, de forma que se salte al código que evalúa $cond_2$ sólo cuando se cumple $cond_1$; cuando $cond_1$ sea falsa, $cond_2$ no debe evaluarse, sino que debe saltarse al mismo sitio que si $cond_2$ fuera falsa, $etq\_2._{Falso}$. De esta manera, si alguna de las condiciones es falsa, el resultado final es que se salta a $etq\_2._{Falso}$, y si las dos son ciertas, entonces se saltará a $etq\_2._{Verdad}$. Por tanto, las etiquetas de $cond_2$ coincidirán con las etiquetas de la condición global, y habrá que colocar acciones semánticas para escribir los **label** tal y como se ha indicado. El diagrama de bloques quedaría tal y como se indica en la figura 8.11.

Si sustituimos las flechas por código de tercetos nos queda un bloque como el de la figura 8.10. Como puede observarse, para generar la etiqueta entre las dos

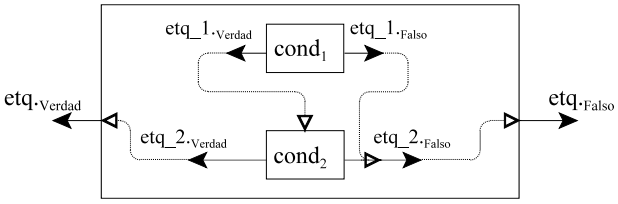


Figura 8.11. Diagrama de bloques asociado a una condición compuesta AND

condiciones será necesario incluir una acción semántica intermedia. Asimismo, también puede apreciarse en esta figura el subterfugio utilizado para conseguir que el destino de falsedad para ambas condiciones sea el mismo: basta con ubicar una sentencia de salto incondicional a la segunda etiqueta justo después del destino de la primera. Además, las etiquetas de verdad y de falso del bloque completo se corresponden con las de verdad y falso de $cond_2$, o sea $etq\_2._{Verdad}$ y $etq\_2._{Falso}$, por lo que ambos atributos deben copiarse tal cual desde $cond_2$ hasta el antecedente cond en la correspondiente regla de producción. La figura 8.12 muestra un ejemplo de código generado para el caso de la condición: a > 3 AND a < 8.

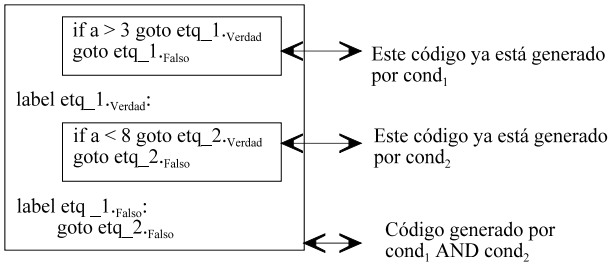


Figura 8.12. Código generado para la condición: a>3 AND a<8

Las acciones semánticas asociadas a la regla de producción interviniente serían:

```
cond :   cond AND      {  printf("label %s\n", $1.etqVerdad);}
           cond        {  printf ("label %s\n", $1.etqFalso);
                          printf ("\tgoto %s\n", $4.etqFalso);
                          strcpy($$.etqVerdad, $4.etqVerdad);
                          strcpy($$.etqFalso, $4.etqFalso);
                }
```

Para comprender bien estas acciones semánticas, debemos recordar que una acción intermedia es traducida por PCYacc a un nuevo no terminal y que, por tanto, cuenta a la hora de numerar los símbolos del consecuente; por ello los atributos asociados a la segunda condición son referidos como $4 y no como $3; $3 es, realmente, la propia acción intermedia. Por otro lado, la acción intermedia podría haberse colocado también justo antes del *token* **AND** habiéndose producido idénticos resultados. No obstante, cuando una acción semántica pueda colocarse en varios puntos de una regla, es conveniente retrasarla lo máximo posible pues con ello pueden evitarse conflictos desplazar/reducir en el analizador sintáctico.

El tratamiento dado a las condiciones compuestas ha sido sistemático y uniforme con respecto a las condiciones simples. Ello nos permite generar grandes bloques de código que, a su vez, pueden intervenir en otras condiciones compuestas. En otras palabras, las acciones semánticas propuestas permiten generar código correctamente para condiciones de la forma: $cond_1$ AND $cond_2$ AND $cond_3$ ... AND $cond_n$.

Como el lector ya ha podido vaticinar, el tratamiento del operador lógico OR obedece a un patrón similar al estudiado para el AND, con la salvedad de que se intercambian los papeles de las etiquetas de verdad y falso en la primera condición, lo que produce un diagrama de bloques como el de la figura 8.13.

La figura 8.14 muestra un ejemplo de código generado para la condición a=3 OR a=5, mientras que las acciones semánticas asociadas a la regla de producción interviniente serían:

```
cond :    cond OR        {  printf("label %s\n", $1.etqFalso);}
               cond       {  printf ("label %s\n", $1.etqVerdad);
                             printf ("\tgoto %s\n", $4.etqVerdad);
                             strcpy($$.etqVerdad, $4.etqVerdad);
                             strcpy($$.etqFalso, $4.etqFalso);
          }
```

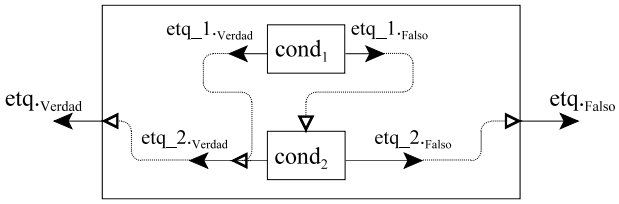


Figura 8.13. Diagrama de bloques asociado a una condición compuesta OR

Por último, el operador NOT tiene un tratamiento muy particular. Si se piensa bien, cuando se reduce por la regla

cond : NOT $cond_1$

el analizador sintáctico habrá ejecutado las acciones semánticas asociadas a la condición del consecuente ($cond_1$), por lo que ya sa habrá generado todo el código necesario para saber si ésta es cierta o no. Por tanto, no tiene sentido generar más código asociado a la aparición del operador NOT, sino que basta con intercambiar los papeles de las etiquetas de verdad y falso del consecuente, que pasarán a ser las

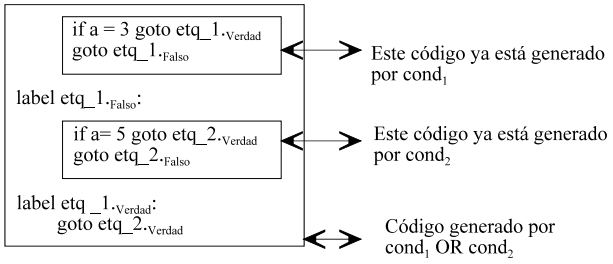


Figura 8.14. Código generado para la condición: a=3 OR a=5

etiquetas de falso y de verdad, respectivamente, del antecedente. En otras palabras, cuando la $cond_1$ es falsa, se produce un salto a la etiqueta de verdad del antecedente, ya que NOT $cond_1$ es cierta.

La figura 8.15 muestra el diagrama de bloques del procedimiento explicado. Por otro lado, la figura 8.16 muestra un ejemplo de los atributos asociados a la condición "a > 3" y a la condición "NOT a > 3", siendo el código generado el mismo en ambos casos.

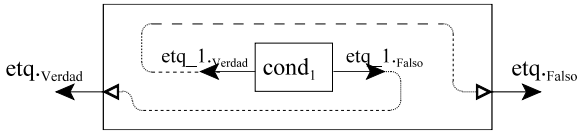


Figura 8.15. Diagrama de bloques asociado a una condición compuesta NOT

La acción semántica asociada a la regla de producción interviniente es:

```
cond : NOT cond {  strcpy($$.etqVerdad, $2.etqFalso);
                   strcpy ($$.etqFalso, $2.etqVerdad);
                }
```

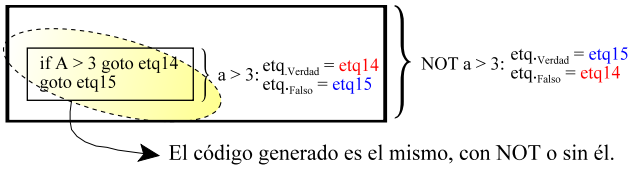


Figura 8.16. Código generado para la condición: NOT a>3

Con esto finalizamos todos los operadores lógicos que posee el lenguaje propuesto. Para otros operadores, tales como NAND, NOR, IMPLIES (implicación lógica), etc., el procedimiento es el mismo. Un caso particular es el del operador XOR, cuya tabla de verdad es:

A	B	A XOR B
Verdad	Verdad	Falso
Verdad	Falso	Verdad
Falso	Verdad	Verdad
Falso	Falso	Falso

Este operador no admite cortocircuito por lo que se debe generar código apoyado por variables temporales para que se salte a una etiqueta de falso caso de que las dos condiciones sean iguales, y a una etiqueta de verdad en caso contrario.

8.4.4 Gestión de sentencias de control de flujo

En este punto abordaremos la utilización de las condiciones y sus etiquetas asociadas, para generar el código de sentencias que alteran el flujo secuencial de ejecución en función del valor de tales condiciones. Estas sentencias son: IF, WHILE, CASE y REPEAT.

8.4.4.1 Sentencia IF-THEN-ELSE

El caso de la sentencia **IF** es el más simple. Aquí basta con indicar que la etiqueta de verdad de la condición está asociada al código a continuación del **THEN**, y la etiqueta de falso se asocia al código que puede haber tras el **ELSE**. En cualquier caso, una vez acabadas las sentencias del **THEN** se debe producir un salto al final del **IF**, porque no queremos que se ejecuten también las sentencias del **ELSE**. Por tanto, tras la sentencias del **THEN**, creamos una nueva etiqueta a la cual produciremos un salto, y colocamos el destino de tal etiqueta al final del código del **IF**. Esto puede apreciarse mejor con un ejemplo:

Código fuente	Código intermedio

IF A > 0 THEN
  S1 := 1;
ELSE
  S2 := 2;
FIN IF
	if A>0 goto etq1
	goto etq2
	label etq1:
	S1 = 1
	goto etq3
	label etq2:
	S2 = 2
	label etq3:

El diagrama de bloques puede apreciarse en la figura 8.17.

Grosso modo, para hacer esto se necesita insertar dos acciones semánticas intermedias, una que visualice el código marcado en rojo y otra para el código marcado en azul. La etiqueta **etq3** se almacena en una variable a la que llamaremos etqFINIF; así, las acciones semánticas necesarias quedarían como se indica a continuación (en este sencilla a proximación se supone que todo **IF** tiene un **ELSE**; más adelante se propondrá el caso general):

```
                        { printf("label %s\n", $2.etqVerdad); }
                        ↓
sent : IF cond THEN   sent ELSE   sent FIN IF { printf("label %s:\n", etqFINIF); }
                                              ↑
                                              { nuevaEtq(etqFINIF);
                                                printf ("\tgoto %s\n", etqFINIF);
                                                printf ("label %s:\n", $2.etqFalso); }
```

Sin embargo, aquí aparece un problema que se convertirá en una constante en el resto de sentencias de control: ¿dónde se declara la variable etqFINIF? (Nótese que esta variable se ha escrito en cursiva en el ejemplo anterior debido a que aún no se ha declarado). Es evidente que cada **IF** distinto que aparezca en el programa fuente

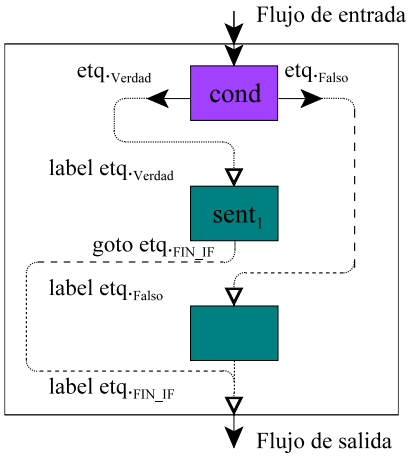


Figura 8.17. Diagrama de bloques de una sentencia **IF**

necesitará una etiqueta de fin de **IF** diferente. Por tanto, está claro que esta etiqueta no debe almacenarse en una variable global ya que la sentencia que hay en el **ELSE** podría ser otro **IF**, lo que haría necesario reducirlo por esta misma regla y machacaría el valor de la variable global antes de llegar a la acción semántica del final (en negro).

Por otro lado, etqFINIF tampoco puede declararse en la acción semántica en azul (donde se le da valor pasándola como parámetro a la función nuevaEtq), ya que en tal caso se trataría de una variable local a un bloque de lenguaje C y no sería visible en la acción semántica del final.

Por tanto, necesitar declararla en algún lugar tal que sea lo bastante global como para ser vista en dos acciones semánticas de la misma regla; pero también debe ser lo bastante local como para que los **IF** anidados puedan asignar valores locales sin machacar las etiquetas de los **IF** más externos. Es de notar que un metacompilador como JavaCC soluciona este problema mediante el área de código asociado a cada regla BNF; aquí se podría declarar la variable etqFINIF y utilizarla en cualquier acción semántica de esa misma regla. En caso de haber **IF** anidados, la propia recursión en las llamadas a las funciones que representan los no terminales se encargaría de crear nuevos ámbitos con copias de dicha variable.

La solución adoptada por JavaCC nos puede dar alguna pista para solucionar el problema en PCYacc: necesitamos una variable local a la regla completa, que se pueda utilizar en todas sus acciones semánticas, pero que sea diferente si hay **IF** anidados. La solución consiste en utilizar el *token* **IF** como contenedor de dicha variable, asignándosela como atributo: el atributo del *token* **IF** es accesible por las acciones semánticas que hay tras él, y si hay, por ejemplo, tres **IF** anidados, cada uno de ellos tendrá su propio atributo. Asumiendo esto, la regla queda:

```
sent : IF cond THEN        { printf("label %s\n", $2.etqVerdad); }
       sent                { nuevaEtq($1);
```

```
                        printf ("\tgoto %s\n", $1);
                        printf ("label %s:\n", $2.etqFalso); }
opcional  FIN IF    { printf("label %s:\n", $1); }
```

La figura 8.18 muestra el código generado para una sentencia de ejemplo. Nótese que si la parte **ELSE** no existe (el no terminal **opcional** se reduce por la regla épsilon), el código funciona correctamente, a pesar de no ser del todo óptimo.

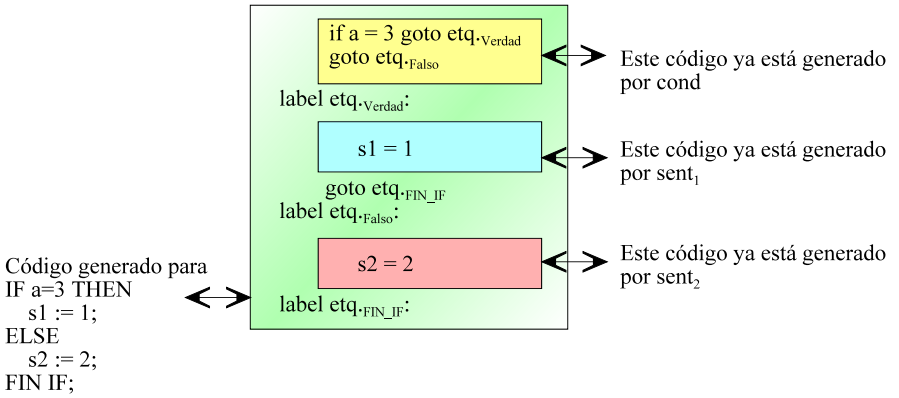


Figura 8.18. Código generado para una sentencia **IF** simple

8.4.4.2 Sentencia WHILE

El caso de **WHILE** y **REPEAT** es muy similar. En ambos es necesario colocar una etiqueta al comienzo del bucle, a la que se saltará para repetir cada iteración.

En el caso de **WHILE**, a continuación se genera el código de la condición, ya que ésta debe evaluarse antes de entrar a ejecutar el cuerpo del bucle. La etiqueta de verdad se ubica justo antes de las sentencias que componen el cuerpo, que es lo que se debe ejecutar si la condición es cierta. Al final de las sentencias se pondrá un salto al inicio del bucle, donde de nuevo se comprobará la condición. La etiqueta de falso de la condición, se ubicará al final de todo lo relacionado con el **WHILE**, o lo que es lo mismo, al principio del código generado para las sentencias que siguen al **WHILE**. De esta manera, cuando la condición deje de cumplirse continuará la ejecución de la secuencia de instrucciones que siguen a la sentencia **WHILE**.

Recordemos que la regla de producción asociada al **WHILE** es:

sent : WHILE cond DO sent FIN WHILE

por lo que los bloques de código de que disponemos para construir adecuadamente nuestro flujo son los que aparecen sombreados en la figura 8.19. El resto de la figura 8.19 muestra dónde deben ubicarse las etiquetas de la condición, así como la necesidad de crear una nueva etiqueta asociada al inicio del **WHILE**. Esta etiqueta tiene

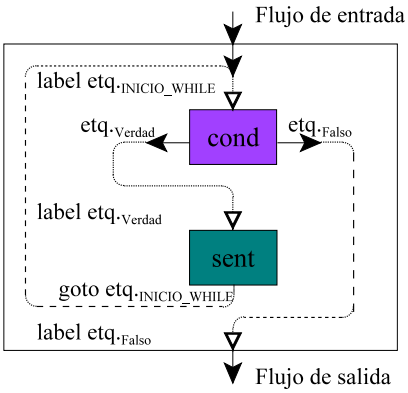


Figura 8.19. Diagrama de bloques de una sentencia **WHILE**

propiedades muy similares a las ya explicadas en el punto anterior con respecto a la sentencia **IF**. En otras palabras, se almacenará como atributo del propio *token* **WHILE**.

De esta forma, el diagrama de bloques sigue exactamente la semántica de un bucle **WHILE**:

- Se comprueba la condición.
- Si es verdad se ejecuta $sent_1$ y se vuelve al punto anterior, a comprobar la condición.
- Si es falsa se sale del bucle y continúa el flujo por la sentencia que sigue al WHILE.

Finalmente, las acciones semánticas asociadas a la regla de producción del **WHILE** quedarían:

```
sent : WHILE                {
                              nuevaEtq($1);
                              printf("label %s:\n", $1);
                            }
       cond DO              { printf ("label %s:\n", $3.etqVerdad); }
       sent FIN WHILE       {
                              printf("\tgoto %s\n", $1);
                              printf("label %s:\n",$3.etqFalso);
                            }
```

8.4.4.3 Sentencia REPEAT

Recordemos la regla de producción asociada a la sentencia **REPEAT**:

```
sent : REPEAT sent UNTIL cond
```

en la que puede observarse cómo la condición se evalúa tras haber ejecutado, al menos, un ciclo. El cuerpo se ejecutará mientras la condición sea falsa, o lo que es lo mismo, hasta que la condición se cumpla.

Para que el código intermedio generado se comporte de la misma manera, lo

ideal es que la etiqueta de falso de la condición se coloque al comienzo del cuerpo, y que la etiqueta de cierto permita continuar el flujo secuencial del programa.

Así, a continuación de la etiqueta de falso se generaría el código de las sentencias, ya que la condición se evalúa al final. Tras las sentencias colocamos el código de la condición. Como la etiqueta de falso ya se ha puesto al comienzo del bucle, sólo queda poner la etiqueta de verdad tras el código de la condición, lo que dará un diagrama de bloques ideal como el que se presenta en la figura 8.20.

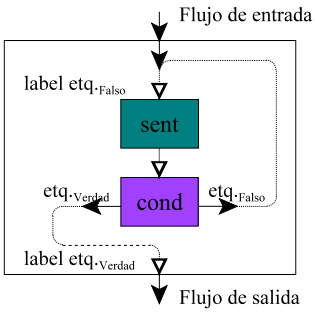


Figura 8.20. Diagrama de bloques ideal de una sentencia REPEAT

Sin embargo, conseguir generar código que obedezca al diagrama de la figura 8.20 Resulta inviable, toda vez que la etiqueta de falso no se puede colocar antes de las sentencias del cuerpo ya que aún no se conoce su valor. En otras palabras, la etiqueta de falso se genera cuando se reduce la condición, cosa que sucede después de que necesitemos visualizar su **label**. Desde otro punto de vista, cuando se llega a conocer el valor de la etiqueta de falso, es tarde para visualizar su **label**, ya que ello tuvo que hacerse incluso antes de reducir las sentencias del cuerpo. Nótese cómo la siguiente acción es incorrecta porque se intenta acceder a un atributo que se desconoce en el momento de ejecutar dicha acción:

```
sent : REPEAT              { printf ("label %s:\n", $5.etqFalso); }
       sent UNTIL cond
```

Para solucionar este problema hacemos lo que se llama una indirección, o sea, creamos una nueva etiqueta de inicio del **REPEAT** y luego creamos un bloque de código en el que colocamos en secuencia:

- El label de la etiqueta de falso.
- Un goto a la etiqueta de inicio del REPEAT.

El diagrama de bloques quedaría como se ilustra en la figura 8.21. Este código tan ineficiente se podría optimizar en una fase posterior.

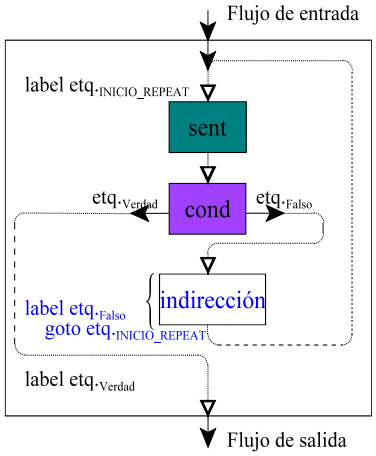


Figura 8.21. Diagrama de bloques de una sentencia REPEAT

Las acciones semánticas asociadas a la regla de producción del **REPEAT** quedarían:

```
sent : REPEAT                {
                               nuevaEtq($1);
                               printf ("label %s:\n", $1);
                             }
       sent UNTIL cond       {
                               printf ("label %s:\n", $5.etqFalso");
                               printf("\tgoto %s\n", $1);
                               printf ("label %s:\n", $5.etqVerdad);

                             }
```

La figura 8.22 muestra el código generado para una sentencia de ejemplo.

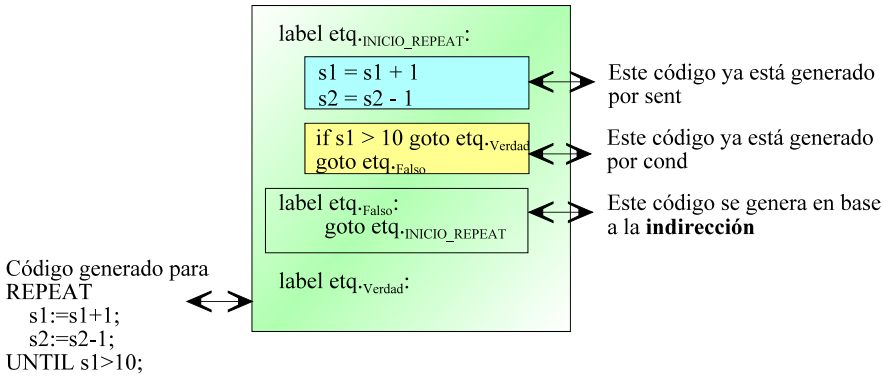


Figura 8.22. Código generado para una sentencia **REPEAT** simple

8.4.4.4 Sentencia CASE

La sentencia **CASE** es la más compleja de traducir, ya que ésta permite un número indeterminado y potencialmente infinito de cláusulas **CASO** en su interior, lo cual obliga a arrastrar una serie de atributos a medida que se van efectuando reducciones. El objetivo semántico de esta sentencia es el mismo que el de la sentencia **switch** del lenguaje C, sólo que no es necesaria una cláusula **break** para romper el flujo al final de la sentencia asociada a cada caso particular.

Por otro lado, y desde el punto de vista gramatical, no podemos utilizar una única regla de producción para indicar su sintaxis por lo que hay que recurrir a reglas recursivas a izquierda y declarar unos cuantos no terminales nuevos. La parte de la gramática que la reconoce es:

```
sent          :  sent_case ;
sent_case     :  inicio_case FIN CASE
              |  inicio_case OTHERWISE sent FIN CASE
              ;
inicio_case   :  CASE expr OF
              |  inicio_case CASO expr ':' sent
              ;
```

Como puede observarse se han incluido dos posibilidades para el no terminal **sent_case**, una con la cláusula **OTHERWISE** y otra sin ella, ya que dicha cláusula es opcional. Ambas reglas podrían haberse sustituído por una sola en la que apareciese un no terminal nuevo que hiciera referencia a la opcionalidad del **OTHERWISE**, como por ejemplo:

```
sent_case       :  inicio_case opcional_case FIN CASE ;
opcional_case   :  OTHERWISE sent
                |  /* Épsilon */
                ;
```

Pero la dejaremos tal y como se ha propuesto para mostrar, junto a la decisión tomada para el IF, cómo una misma cosa se puede hacer de diferentes formas y cómo afecta cada decisión a la hora de incluir las acciones semánticas para generar código intermedio.

Además, la gramática propuesta permite construcciones como "CASE a OF FIN CASE", en la que no se ha incluído ni una sóla cláusula **CASO**. Desde nuestro punto de vista lo consideraremos válido, pero si quisiéramos evitarlo tendríamos que sustituir la primera de **inicio_case** y ponerla como:

```
inicio_case : CASE expr OF CASO expr ':' sent
```

La figura 8.23 muestra parte del árbol sintáctico que reconoce una sentencia **CASE**, en la que se puede apreciar la recursión sobre el no terminal **inicio_case**.

Una vez vistos estos detalles, vamos a pasar ahora a la generación de código intermedio en sí. De lo primero que hay que percatarse es de que la regla principal que

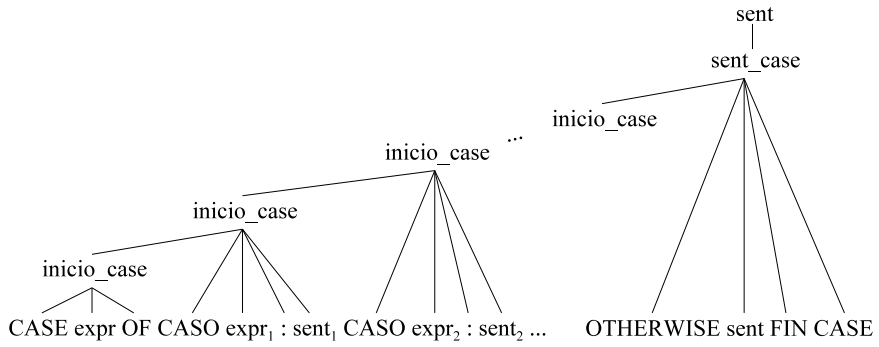


Figura 8.23. Árbol sintáctico que reconoce una sentencia **CASE**

se va a encargar de generar código es la que más aparece en el árbol sintáctico, esto es, la regla recursiva de **inicio_case**:

inicio_case : inicio_case CASO expr ':' sent

de tal manera que la/s acciones semánticas que asociemos a esta regla se ejecutarán repetidas veces durante la reducción de la sentencia **CASE** completa. Teniendo esto en cuenta vamos a dar una idea preliminar de la estructura del código intermedio que queremos generar, considerando que la estructura que controla cada bloque **CASO** debe ser siempre la misma (la estructura, pero no las variables y etiquetas que intervienen en cada **CASO**). Así pues, asociado al ejemplo de la figura 8.23 se tendrían unos bloques como los de la figura 8.24. En esta última figura se han marcado en diferentes color los bloques de código en función de qué regla de producción lo ha generado. En azul están los bloques generados por la segunda regla de **inicio_case**, en verde el generado por la primera regla de **inicio_case**, y en marrón el generado por la regla de **sent_case** que incluye la cláusula **OTHERWISE**. Asimismo, al margen de cada bloque **expr** figura el nombre de la variable (temporal o no) en la que se asume que se almacena el resultado total de evaluar dicha expresión en tiempo de ejecución. Por último, los bloques rayados representan código

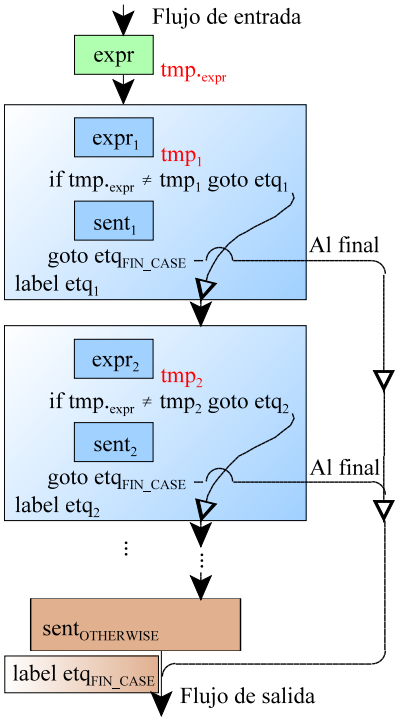


Figura 8.24. Bloques de código para una sentencia **CASE**

intermedio que ya ha sido generado por otras reglas de producción; por tanto, todo lo que no está rayado debemos generarlo en acciones semánticas asociadas a las reglas de producción del **CASE**.

Como puede verse en esta figura, se ha conseguido que todos los bloques en azul tengan aproximadamente la misma forma:

- el cálculo de una expresión local que se almacena en $\mathbf{tmp_i}$
- una comparación de $\mathbf{tmp._{expr}}$ con $\mathbf{tmp_i}$. Si no son iguales saltamos al final del bloque
- el código intermedio asociado a una sentencia $\mathbf{sent_i}$
- un salto al final de la sentencia **CASE**

Parece claro que la mayor complejidad en la generación de este código, radica en la regla del **CASO** (segunda regla de **inicio_case**), ya que ella es la que debe generar todo el código marcado de azul en la figura 8.24. Estudiemos con detenimiento uno de los bloques azules. Los tercetos que debemos generar en cada bloque de código azul se dividen, a su vez en dos partes: una intercalada entre $\mathbf{expr_i}$ y $\mathbf{sent_i}$, y otra al final del bloque. El siguiente paso a seguir es examinar este código para deducir qué atributos vamos a necesitar en cada no terminal. Centrándonos en las variables y etiquetas que aparecen, podemos clasificarlas en dos apartados:

- La variable $\mathbf{tmp_i}$ y la etiqueta $\mathbf{etq_i}$ son locales a cada bloque, esto es, no se utilizan más que en un solo bloque.
- La variable $\mathbf{tmp._{expr}}$ y la etiqueta $\mathbf{etq_{FIN\_CASE}}$ se utilizan repetidamente en todos los bloques. Además se utiliza en la regla de **sent_case** con el objetivo de visualizar el "**label** $\mathbf{etq_{FIN\_CASE}}$" último. $\mathbf{tmp._{expr}}$ representa a la expresión que se va a ir comparando y $\mathbf{etq_{FIN\_CASE}}$ representa el final del **CASE**, etiqueta a la que se saltará tras poner el código asociado a la sentencia de cada **CASO**.

De un lado, la variable $\mathbf{tmp_i}$ es generada por la expresión local al **CASO** y almacenada en el atributo del no terminal **expr**, por lo que es accesible en cualquier acción semántica de la regla de producción que se ponga detrás de dicho no terminal. En cuanto a la variable $\mathbf{etq_i}$ puede observarse que se utiliza en un **goto** en un terceto entre la generación del código de $\mathbf{expr_i}$ y de $\mathbf{sent_i}$, mientras que el **label** aparece después de $\mathbf{sent_i}$. Esto quiere decir que necesitaremos esta etiqueta en dos acciones semánticas de la misma regla, por lo que optaremos por asociarla como atributo del *token* **CASO**, de forma parecida a como se ha hecho para las sentencias **IF**, **WHILE** y **REPEAT**.

De otro lado, y un poco más complicado, la variable $\mathbf{tmp._{expr}}$ se genera por la **expr** que aparece en la regla del **CASE**, pero debe utilizarse en todas y cada una de las reglas de **CASO** con el objetivo de comparar su valor con las sucesivas $\mathbf{tmp_i}$. Por ello hemos de encontrar algún mecanismo que permita propagar el nombre de esta variable a través de todas las reducciones de la regla del **CASO** necesarias para reconocer la sentencia completa. Con la etiqueta $\mathbf{etq_{FIN\_CASE}}$ sucede algo parecido, ya que dicha etiqueta es necesaria en cada bloque **CASO** con el objetivo de saltar al final del **CASE**

una vez finalizada la ejecución de la sentencia asociada a cada **CASO**. Asimismo, $\mathbf{etq_{FIN\_CASE}}$ debe ser accesible por la regla **sent_case** para colocar el **label** final (en marrón según la figura 8.24).

Para solucionar la accesibilidad de $\mathbf{tmp._{expr}}$ y $\mathbf{etq_{FIN\_CASE}}$ por las acciones semánticas que las necesitan, lo mejor es asociarlas como atributos del no terminal **inicio_case**. Esta decisión es crucial, y supone el mecanismo central por el que daremos solución a la generación de código para la sentencia **CASE**. De esta forma, y tal y como puede apreciarse en la figura 8.25, ambas variables deben propagarse de un no terminal **inicio_case** a otro, haciendo uso de la regla recursiva:

```
inicio_case    :    inicio_case CASO expr ':' sent
```

Así, las acciones semánticas de esta regla deben generar el código intermedio en base a los atributos del **inicio_case** del consecuente y, por último, propagar estos atributos al **inicio_case** del antecedente con el objetivo de que puedan ser utilizados en el siguiente **CASO** (si lo hay). Como puede observarse, el encargado de generar la $\mathbf{etq_{FIN\_CASE}}$ es la primera regla de **inicio_case**, a pesar de que esta regla no va a necesitar dicho atributo: el objetivo es que todas las demás aplicaciones de la regla recursiva dispongan de la misma etiqueta a la que saltar.

Así pues, las reglas de producción y sus acciones semánticas quedan, comenzando por las que primero se reducen:

```
inicio_ case    :  CASE expr OF    {
                                     strcpy ($$.var,$2);
                                     nuevaEtq ($$.etq );
                                   }
          |  inicio_case CASO expr ':'  {
                                          nuevaEtq( $2.etq);
                                          printf ("if %s != %s goto %s",
                                                       $1.var, $3, $2.etq);
                                        }
```

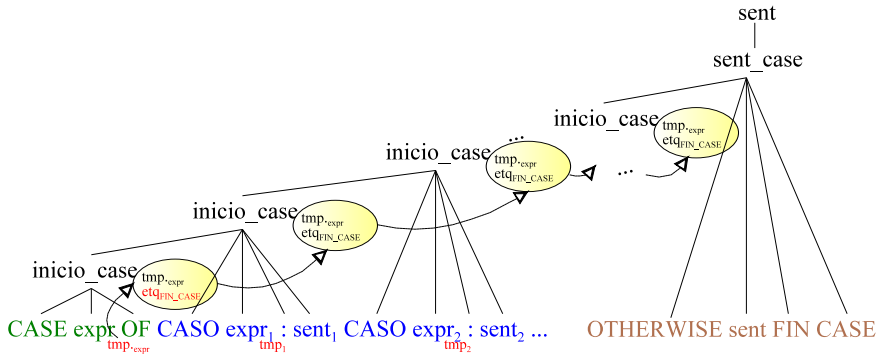


Figura 8.25. Propagación de atributos del no terminal **inicio_case**. Las cláusulas se han coloreado de acuerdo a la figura 8.24. En rojo se ha indicado la primera aparición de cada atributo

```
                        sent            {
                                          strcpy ($$.var, $1.var);
                                          strcpy( $$.etq, $1.etq);
                                          printf("goto %s", $2.etq);
                                          printf("label %s", $2.etq);
                                        }
            ;
sent_ case  : inicio_case OTHERWISE sent FIN CASE { printf("label %s", $1.etq); }
            | inicio_case FIN CASE { printf("label %s", $1.etq); }
            ;
```

8.4.5 Solución con Lex/Yacc

A continuación se expone todo el código explicado en los apartados anteriores. Comenzamos con el código Lex cuyo único objetivo es reconocer las palabras reservadas, los identificadores y los números, así como ignorar comentarios y llevar la cuenta del número de línea por el que avanza el análisis léxico. Los comentarios se extienden en líneas completas y comienzan por un asterisco:

```
1  %{
2        int linea_actual = 1;
3  %}
4  %START COMENT
5  %%
6  ^[ \t]*"*"        { BEGIN COMENT; }
7  <COMENT>.+     { ; }
8  <COMENT>\n     { BEGIN 0; linea_actual ++; }
9
10 ":="       { return ASIG; }
11 ">="       { return MAI; }
12 "<="       { return MEI; }
13 "!="       { return DIF; }
14 CASE       { return CASE; }
15 OF         { return OF; }
16 CASO       { return CASO; }
17 OTHERWISE      { return OTHERWISE; }
18 REPEAT       { return REPEAT; }
19 UNTIL      { return UNTIL; }
20 IF         { return IF; }
21 THEN       { return THEN; }
22 ELSE       { return ELSE; }
23 WHILE      { return WHILE; }
24 DO         { return DO; }
25 AND        { return AND; }
26 OR         { return OR; }
27 NOT        { return NOT; }
28 FIN        { return FIN; }
29
30 [0-9]+     {
31                strcpy(yylval.numero, yytext);
```

```
32                  return NUMERO;
33            }
34 [A-Za-z_][A-Za-z0-9_]*      {
35                                    strcpy(yylval.variable_aux, yytext);
36                                    return ID;
37                                }
38 [ \t]+       { ; }
39 \n           { linea_actual++; }
40 .            { return yytext[0]; }
```

En cuanto al código Yacc, es el siguiente:

```
 1 /* Declaraciones de apoyo */
 2 %{
 3 typedef struct _doble_cond {
 4       char      etq_verdad[21],
 5                 etq_falso[21];
 6 } doble_cond;
 7 typedef struct _datos_case {
 8       char      etq_final[21];
 9       char      variable_expr[21];
10 } datos_case;
11 %}
12 /* Declaracion de atributos */
13 %union {
14       char numero[21];
15       char variable_aux[21];
16       char etiqueta_aux[21];
17       char etiqueta_siguiente[21];
18       doble_cond bloque_cond;
19       datos_case bloque_case;
20 }
21 /* Declaracion de tokens y sus atributos */
22 %token <numero> NUMERO
23 %token <variable_aux> ID
24 %token <etiqueta_aux> IF WHILE REPEAT
25 %token <etiqueta_siguiente> CASO
26 %token ASIG THEN ELSE FIN DO UNTIL CASE OF OTHERWISE
27 %token MAI MEI DIF
28 /* Declaración de no terminales y sus atributos */
29 %type <variable_aux> expr
30 %type <bloque_cond> cond
31 %type <bloque_case> inicio_case
32 /* Precedencia y asociatividad de operadores */
33 %left OR
34 %left AND
35 %left NOT
36 %left '+' '-'
37 %left '*' '/'
38 %left MENOS_UNARIO
39
```

```
40 %%
41 prog      :     prog sent ';'
42           |     prog error ';' { yyerrok; }
43           |
44           ;
45 sent      :     ID ASIG expr      {
46                                       printf("\t%s = %s\n", $1, $3);
47                                     }
48           |     IF cond {
49                           printf("label %s\n", $2.etq_verdad);
50                         }
51                 THEN sent ';' {
52                                 nuevaEtq($1);
53                                 printf("\tgoto %s\n", $1);
54                                 printf("label %s\n", $2.etq_falso);
55                               }
56                 opcional
57                 FIN IF    {
58                           printf("label %s\n", $1);
59                         }
60           |     '{' lista_sent '}' { ; }
61           |     WHILE  {
62                           nuevaEtq($1);
63                           printf("label %s\n", $1);
64                         }
65                 cond      {
66                           printf("label %s\n", $3.etq_verdad);
67                         }
68                 DO sent ';'
69                 FIN WHILE  {
70                                 printf("\tgoto %s\n", $1);
71                                 printf("label %s\n", $3.etq_falso);
72                               }
73           |     REPEAT      {
74                                 nuevaEtq($1);
75                                 printf("label %s\n", $1);
76                               }
77                 sent ';'
78                 UNTIL cond  {
79                                 printf("label %s\n", $6.etq_falso);
80                                 printf("\tgoto %s\n", $1);
81                                 printf("label %s\n", $6.etq_verdad);
82                               }
83           |     sent_case
84           ;
85 opcional :     ELSE sent ';'
86                 |     /* Epsilon */
87                 ;
88 lista_sent      :     /* Epsilon */
```

```
89                  |   lista_sent sent ';'
90                  |   lista_sent error ';' { yyerrok; }
91                  ;
92  sent_case       :   inicio_case
93                      OTHERWISE sent ';'
94                      FIN CASE {
95                                  printf("label %s\n", $1.etq_final);
96                              }
97                  |   inicio_case
98                      FIN CASE {
99                                  printf("label %s\n", $1.etq_final);
100                             }
101                 ;
102 inicio_case    :   CASE expr OF    {
103                                       strcpy($$.variable_expr, $2);
104                                       nuevaEtq($$.etq_final);
105                                     }
106                 |   inicio_case
107                     CASO expr ':' {
108                                     nuevaEtq($2);
109                                     printf("\tif %s != %s goto %s\n",
110                                             $1.variable_expr, $3, $2);
111                                   }
112                     sent ';'      {
113                                     printf("\tgoto %s\n", $1.etq_final);
114                                     printf("label %s\n", $2);
115                                     strcpy($$.variable_expr, $1.variable_expr);
116                                     strcpy($$.etq_final, $1.etq_final);
117                                   }
118                 ;
119 expr        :   NUMERO      { generarTerceto("\t%s = %s\n", $$, $1, NULL); }
120             |   ID          { strcpy($$, $1); }
121             |   expr '+' expr { generarTerceto("\t%s = %s + %s\n", $$, $1, $3); }
122             |   expr '-' expr  { generarTerceto("\t%s = %s - %s\n", $$, $1, $3); }
123             |   expr '*' expr  { generarTerceto("\t%s = %s * %s\n", $$, $1, $3); }
124             |   expr '/' expr  { generarTerceto("\t%s = %s / %s\n", $$, $1, $3); }
125             |   '-' expr %prec MENOS_UNARIO
126                             { generarTerceto("\t%s = - %s\n", $$, $2, NULL); }
127             |   '(' expr ')'  { strcpy($$, $2); }
128             ;
129 cond        :   expr '>' expr       { generarCondicion($1, ">", $3, &($$)); }
130             |   expr '<' expr       { generarCondicion($1, "<", $3, &($$)); }
131             |   expr MAI expr       { generarCondicion($1, ">=", $3, &($$)); }
132             |   expr MEI expr       { generarCondicion($1, "<=", $3, &($$)); }
133             |   expr '=' expr       { generarCondicion($1, "=", $3, &($$)); }
134             |   expr DIF expr       { generarCondicion($1, "!=", $3, &($$)); }
135             |   NOT cond            { strcpy($$.etq_verdad, $2.etq_falso);
136                                       strcpy($$.etq_falso, $2.etq_verdad);
137                                     }
```

```
138         |   cond AND    {
139                             printf("label %s\n", $1.etq_verdad);
140                           }
141             cond          {
142                             printf("label %s\n", $1.etq_falso);
143                             printf("\tgoto %s\n", $4.etq_falso);
144                             strcpy($$.etq_verdad, $4.etq_verdad);
145                             strcpy($$.etq_falso, $4.etq_falso);
146                           }
147         |   cond OR     {
148                             printf("label %s\n", $1.etq_falso);
149                           }
150             cond          {
151                             printf("label %s\n", $1.etq_verdad);
152                             printf("\tgoto %s\n", $4.etq_verdad);
153                             strcpy($$.etq_verdad, $4.etq_verdad);
154                             strcpy($$.etq_falso, $4.etq_falso);
155                           }
156         |   '(' cond ')'    {
157                             strcpy($$.etq_verdad, $2.etq_verdad);
158                             strcpy($$.etq_falso, $2.etq_falso);
159                           }
160         ;
161
162 %%
163 #include "ejem6l.c"
164 void main() {
165    yyparse();
166 }
167 void yyerror(char * s) {
168    fprintf(stderr, "Error de sintaxis en la linea %d\n", linea_actual);
169 }
170 void nuevaTmp(char * s) {
171   static actual=0;
172    sprintf(s, "tmp%d", ++actual);
173 }
174 void nuevaEtq(char * s) {
175   static actual=0;
176    sprintf(s, "etq%d", ++actual);
177 }
178 void generarTerceto(      char * terceto,
179                   char * Lvalor,
180                   char * Rvalor1,
181                   char * Rvalor2){
182       nuevaTmp(Lvalor);
183       printf(terceto, Lvalor, Rvalor1, Rvalor2);
184 }
185 void generarCondicion(   char * Rvalor1,
186                   char * condicion,
```

```
187                 char * Rvalor2,
188                 doble_cond * etqs){
189         nuevaEtq((*etqs).etq_verdad);
190         nuevaEtq((*etqs).etq_falso);
191         printf("\tif %s %s %s goto %s\n", Rvalor1, condicion, Rvalor2, (*etqs).etq_verdad);
192         printf("\tgoto %s\n", (*etqs).etq_falso);
193 }
```

Para empezar, la generación de los tercetos más frecuentes (expresiones aritméticas y condiciones simples) se ha delegado en las funciones generarTerceto y GenerarCondicion de las líneas 178 a 193, que reciben los parámetros adecuados y hacen que las reglas de las líneas 119 a 134 queden mucho más compactas y claras.

El %union de la línea 13 junto a las declaraciones de tipos de las líneas precedentes establece el marco de atributos para los terminales y no terminales de la gramática. Es de notar que el %union posee varios campos con el mismo tipo (char[21]), pero distinto nombre: numero, variable_aux, etiqueta_aux y etiqueta_siguiente, al igual que dos registros con la misma estructura interna (dos cadenas de 21 caracteres): bloque_cond y bloque_case. Ello se debe a que, aunque estructuralmente coincidan, semánticamente tienen objetivos distintos, por lo que se ha optado por replicarlos con nombres distintivos. Esto no tiene ninguna repercusión desde el punto de vista de la eficiencia, puesto que el espacio ocupado por el %union es el mismo.

Por último, como atributo del *token* **NUMERO** se ha escogido un campo de tipo texto en lugar de un valor numérico entero, ya que nuestro propósito es generar tercetos y no operar aritméticamente con los valores en sí. Además, la acción semántica de la línea 119 obliga a que todo literal numérico sea gestionado mediante una variable temporal merced a un terceto de asignación directa.

8.4.6 Solución con JFlex/Cup

La solución con JFlex y Cup es muy similar a la dada anteriormente para Lex y Yacc. La única diferencia relevante con respecto a JFlex radica en que debe ser éste quien se encargue de asignar las etiquetas necesarias a los terminales **IF**, **WHILE**, **REPEAT** y **CASO** debido a que los atributos no se pueden modificar en las acciones intermedias de Cup. Para dejar esto más patente el código JFlex tiene una función estática que permite generar etiquetas (con el prefijo **etqL** para distinguirlas de las generadas por Cup que llevarán el prefijo **etqY**), ya que las acciones de JFlex no pueden acceder a las definiciones de funciones declaradas en Cup. Así pues el código queda:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  %%
4  %{
5       int lineaActual = 1;
6       private static int actualEtq=0;
7       private static String nuevaEtq() {
```

```
8               return "etqL"+(++actualEtq);
9           }
10  %}
11  %unicode
12  %cup
13  %line
14  %column
15  %state COMENT
16  %%
17  ^[ \t]*"*"                  { yybegin(COMENT); }
18  <COMENT>.+      { ; }
19  <COMENT>\n      { lineaActual ++; yybegin(YYINITIAL); }
20  "+"   { return new Symbol(sym.MAS); }
21  "*"   { return new Symbol(sym.POR); }
22  "/"   { return new Symbol(sym.ENTRE); }
23  "-"   { return new Symbol(sym.MENOS); }
24  "("   { return new Symbol(sym.LPAREN); }
25  ")"   { return new Symbol(sym.RPAREN); }
26  "{"   { return new Symbol(sym.LLLAVE); }
27  "}"   { return new Symbol(sym.RLLAVE); }
28  ";"   { return new Symbol(sym.PUNTOYCOMA); }
29  ":"   { return new Symbol(sym.DOSPUNTOS); }
30  ":="  { return new Symbol(sym.ASIG); }
31  ">"   { return new Symbol(sym.MAYOR); }
32  "<"   { return new Symbol(sym.MENOR); }
33  "="   { return new Symbol(sym.IGUAL); }
34  ">="        { return new Symbol(sym.MAI); }
35  "<="        { return new Symbol(sym.MEI); }
36  "!="        { return new Symbol(sym.DIF); }
37  CASE    { return new Symbol(sym.CASE); }
38  OF      { return new Symbol(sym.OF); }
39  CASO    { return new Symbol(sym.CASO, nuevaEtq()); }
40  OTHERWISE       { return new Symbol(sym.OTHERWISE); }
41  REPEAT          { return new Symbol(sym.REPEAT, nuevaEtq()); }
42  UNTIL   { return new Symbol(sym.UNTIL); }
43  IF      { return new Symbol(sym.IF, nuevaEtq()); }
44  THEN    { return new Symbol(sym.THEN); }
45  ELSE    { return new Symbol(sym.ELSE); }
46  WHILE   { return new Symbol(sym.WHILE, nuevaEtq()); }
47  DO      { return new Symbol(sym.DO); }
48  AND     { return new Symbol(sym.AND); }
49  OR      { return new Symbol(sym.OR); }
50  NOT     { return new Symbol(sym.NOT); }
51  FIN     { return new Symbol(sym.FIN); }
52  [:jletter:][:jletterdigit:]*  { return new Symbol(sym.ID, yytext()); }
53  [:digit:]+                    { return new Symbol(sym.NUMERO, yytext()); }
54  [ \t\r]+    {;}
55  [\n]        { lineaActual++; }
56  .           { System.out.println("Error léxico en línea "+lineaActual+":-"+yytext()+"-"); }
```

Con respecto a Cup, la diferencia fundamental radica en la utilización de objetos pertenecientes a las clases DatosCASE y BloqueCondicion para poder asignar más de un atributo a algunos no terminales. Por otro lado, también es de notar que la funciones generaTerceto y generaCondicion se han adaptado a la filosofía Java y, a diferencia de Lex y Yacc, ahora generan un atributo que se debe asignar a RESULT en la invocación. Por último, los nombres de atributos *in situ* (indicados en la propia regla y separados del símbolo a que pertenecen mediante dos puntos) han sido escogidos con cuidado para hacer el código más legible. El programa Cup queda:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  parser code {:
4      public static void main(String[] arg){
5          Ylex miAnalizadorLexico = new Ylex(new InputStreamReader(System.in));
6          parser parserObj = new parser(miAnalizadorLexico);
7          try{
8              parserObj.parse();
9          }catch(Exception x){
10             x.printStackTrace();
11             System.out.println("Error fatal.");
12         }
13     }
14 :};
15 action code {:
16     class DatosCASE {
17         String tmpExpr, etqFinal;
18     }
19     class BloqueCondicion {
20         String etqVerdad, etqFalso;
21     }
22     private static int actualTmp=0;
23     private static String nuevaTmp() {
24         return "tmp"+(++actualTmp);
25     }
26     private static int actualEtq=0;
27     private static String nuevaEtq() {
28         return "etqY"+(++actualEtq);
29     }
30     private String generarTerceto(String terceto) {
31         String tmp = nuevaTmp();
32         System.out.println(tmp + terceto);
33         return tmp;
34     }
35     private BloqueCondicion generarCondicion(  String Rvalor1,
36                                                String condicion,
37                                                String Rvalor2) {
38         BloqueCondicion etqs = new BloqueCondicion();
39         etqs.etqVerdad = nuevaEtq();
40         etqs.etqFalso = nuevaEtq();
```

```
41              System.out.println("\tif "+ Rvalor1 + condicion + Rvalor2
42                                                      +" goto "+ etqs.etqVerdad);
43              System.out.println("\tgoto "+ etqs.etqFalso);
44              return etqs;
45          }
46  :}
47  terminal PUNTOYCOMA, DOSPUNTOS, MAS, POR, ENTRE, MENOS, UMENOS;
48  terminal LPAREN, RPAREN, LLLAVE, RLLAVE;
49  terminal MAYOR, MENOR, IGUAL, MAI, MEI, DIF;
50  terminal AND, OR, NOT;
51  terminal String ID, NUMERO;
52  terminal String IF, WHILE, REPEAT, CASO;
53  terminal ASIG, THEN, ELSE, FIN, DO, UNTIL, CASE, OF, OTHERWISE;
54  non terminal String expr;
55  non terminal BloqueCondicion cond;
56  non terminal DatosCASE inicioCASE;
57  non terminal prog, sent, sentCASE, opcional, listaSent;
58  precedence left OR;
59  precedence left AND;
60  precedence right NOT;
61  precedence left MAS, MENOS;
62  precedence left POR, ENTRE;
63  precedence right UMENOS;
64  /* Gramática */
65  prog      ::=   prog sent PUNTOYCOMA
66              |   prog error PUNTOYCOMA
67              |
68              ;
69  sent      ::=   ID:id ASIG expr:e       {:
70                      System.out.println("\t"+ id +" = "+ e);
71                  :}
72              |   IF:etqFinIf cond:c       {:
73                      System.out.println("label "+ c.etqVerdad);
74                  :}
75                  THEN sent PUNTOYCOMA {:
76                      System.out.println("\tgoto "+ etqFinIf);
77                      System.out.println("label "+ c.etqFalso);
78                  :}
79                  opcional
80                  FIN IF    {:
81                      System.out.println("label "+ etqFinIf);
82                  :}
83
84              |   LLLAVE listaSent RLLAVE {: ; :}
85              |   WHILE:etqInicioWhile        {:
86                      System.out.println("label "+ etqInicioWhile);
87                  :}
88                  cond:c   {:
89                      System.out.println("label "+ c.etqVerdad);
```

```
90                  :}
91              DO sent PUNTOYCOMA
92              FIN WHILE {:
93                    System.out.println("\tgoto "+ etqInicioWhile);
94                    System.out.println("label "+ c.etqFalso);
95                  :}
96          |   REPEAT:etqInicioRepeat  {:
97                    System.out.println("label "+ etqInicioRepeat);
98                  :}
99              sent PUNTOYCOMA
100             UNTIL cond:c {:
101                   System.out.println("label "+ c.etqFalso);
102                   System.out.println("\tgoto "+ etqInicioRepeat);
103                   System.out.println("label "+ c.etqVerdad);
104                 :}
105         |   sentCASE
106         ;
107 opcional ::=  ELSE sent PUNTOYCOMA
108             |   /* Epsilon */
109             ;
110 listaSent   ::=  /* Epsilon */
111             |   listaSent sent PUNTOYCOMA
112             |   listaSent error PUNTOYCOMA
113             ;
114 sentCASE    ::=  inicioCASE:ic OTHERWISE sent PUNTOYCOMA
115                  FIN CASE {:
116                        System.out.println("label "+ ic.etqFinal);
117                       :}
118             |   inicioCASE:ic
119                  FIN CASE {:
120                        System.out.println("label "+ ic.etqFinal);
121                       :}
122             ;
123 inicioCASE  ::=  CASE expr:e OF {:
124                       RESULT = new DatosCASE();
125                       RESULT.tmpExpr = e.nombreVariable;
126                       RESULT.etqFinal = nuevaEtq();
127                      :}
128             |   inicioCASE:ic
129                  CASO:etqFinCaso expr:e DOSPUNTOS {:
130                       System.out.println("\tif "+ ic.tmpExpr +" != "+ e
131                                                    +" goto "+ etqFinCaso);
132                      :}
133                  sent PUNTOYCOMA {:
134                       System.out.println("\tgoto "+ ic.etqFinal);
135                       System.out.println("label "+ etqFinCaso);
136                       RESULT = ic;
137                      :}
138             ;
```

```
139 expr ::= ID:id                  {: RESULT = id; :}
140       |  NUMERO:n               {: RESULT = generarTerceto(" = "+ n); :}
141       |  expr:e1 MAS expr:e2  {: RESULT = generarTerceto(" = "+ e1 +" + "+ e2); :}
142       |  expr:e1 MENOS expr:e2 {: RESULT=generarTerceto(" = "+ e1 +" - "+ e2); :}
143       |  expr:e1 POR expr:e2   {: RESULT = generarTerceto(" = "+ e1 +" * "+ e2); :}
144       |  expr:e1 ENTRE expr:e2 {: RESULT =generarTerceto(" = "+ e1 +" / "+ e2); :}
145       |  MENOS expr:e1         {: RESULT = generarTerceto(" = -"+ e1); :}
146                  %prec UMENOS
147       |  LPAREN expr:e1 RPAREN       {: RESULT = e1; :}
148       ;
149 cond  ::=   expr:e1 MAYOR expr:e2  {: RESULT = generarCondicion(e1, ">", e2); :}
150       |      expr:e1 MENOR expr:e2  {: RESULT = generarCondicion(e1, "<", e2); :}
151       |      expr:e1 MAI expr:e2       {: RESULT = generarCondicion(e1, ">=", e2); :}
152       |      expr:e1 MEI expr:e2       {: RESULT = generarCondicion(e1, "<=", e2); :}
153       |      expr:e1 IGUAL expr:e2   {: RESULT = generarCondicion(e1, "=", e2); :}
154       |      expr:e1 DIF expr:e2       {: RESULT = generarCondicion(e1, "!=", e2); :}
155       |      NOT cond:c     {:
156                              RESULT = new BloqueCondicion();
157                              RESULT.etqVerdad = c.etqFalso;
158                              RESULT.etqFalso = c.etqVerdad;
159                             :}
160       |      cond:c1 AND {:
161                              System.out.println("label "+ c1.etqVerdad);
162                             :}
163              cond:c2     {:
164                              System.out.println("label "+ c1.etqFalso);
165                              System.out.println("\tgoto "+ c2.etqFalso);
166                              RESULT = c2;
167                             :}
168       |      cond:c1 OR      {:
169                              System.out.println("label "+ c1.etqFalso);
170                             :}
171              cond:c2     {:
172                              System.out.println("label "+ c1.etqVerdad);
173                              System.out.println("\tgoto "+ c2.etqVerdad);
174                              RESULT = c2;
175                             :}
176       |      LPAREN cond:c1 RPAREN    {: RESULT = c1; :}
177       ;
```

8.4.7 Solución con JavaCC

La solución con JavaCC es muy similar a la dada en JFlex y Cup, sólo que se aprovechan las características de la notación BNF para ahorrar algunos atributos; por ejemplo, ya no es necesaria la clase **DatosCASE** pues en una sola regla se dispone de toda la información necesaria.

Por desgracia, JavaCC no posee una mínima detección de sensibilidad al contexto en su analizador lexicográfico, por lo que detectar los comentarios al inicio

de la línea pasa por utilizar estado léxicos que complican un poco la notación, y que obligan a declarar todos y cada uno de los *tokens* de la gramática con el objetivo de pasar al estado por defecto (**DEFAULT**) una vez reconocido cada uno de ellos. Esto provoca, además, que haya que comenzar el análisis lexicográfico con el estado léxico INICIO_LINEA activado, lo que se encarga de hacer la función main() a través de la función SwitchTo de la clase TokenManager.

Así, la solución completa es:

```
1  PARSER_BEGIN(Control)
2      import java.util.*;
3      public class Control{
4
5          private static class BloqueCondicion{
6              String etqVerdad, etqFalso;
7          }
8          public static void main(String args[]) throws ParseException {
9              ControlTokenManager tm =
10                     new ControlTokenManager(new SimpleCharStream(System.in));
11             tm.SwitchTo(tm.INICIO_LINEA);
12             new Control(tm).gramatica();
13         }
14         private static int actualTmp=0, actualEtq=0;
15         private static String nuevaTmp(){
16             return "tmp"+(++actualTmp);
17         }
18         private static String nuevaEtq(){
19             return "etq"+(++actualEtq);
20         }
21         private static void usarASIG(String s, String e){
22             System.out.println(s+"="+e);
23         }
24         private static String usarOpAritmetico(String e1, String e2, String op){
25             String tmp = nuevaTmp();
26             System.out.println("\t"+tmp+"="+e1+op+e2);
27             return tmp;
28         }
29         private static void usarLabel(String label){
30             System.out.println("label "+ label);
31         }
32         private static void usarGoto(String label){
33             System.out.println("\tgoto "+ label);
34         }
35         private static BloqueCondicion usarOpRelacional(String e1,
36                                                         String e2,
37                                                         String op){
38             BloqueCondicion blq = new BloqueCondicion();
39             blq.etqVerdad = nuevaEtq();
40             blq.etqFalso = nuevaEtq();
```

```
41                  System.out.println("\tif "+ e1+op+e2 +" goto "+ blq.etqVerdad);
42                  usarGoto(blq.etqFalso);
43                  return blq;
44              }
45              private static void intercambiarCondicion(BloqueCondicion blq){
46                  String aux = blq.etqVerdad;
47                  blq.etqVerdad = blq.etqFalso;
48                  blq.etqFalso = blq.etqVerdad;
49              }
50          }
51  PARSER_END(Control)
52  <DEFAULT, INICIO_LINEA> SKIP : {
53          " "
54      |   "\t"
55      |   "\r"
56      |   "\n" : INICIO_LINEA
57  }
58  <INICIO_LINEA> SKIP: {
59      <COMENTARIO: "*" (~["\n"])* "\n">
60  }
61  <DEFAULT, INICIO_LINEA> TOKEN [IGNORE_CASE] : {
62          <NUMERO: (["0"-"9"])+>    : DEFAULT
63      |   <PUNTOYCOMA: ";">       : DEFAULT
64  }
65  <DEFAULT, INICIO_LINEA> TOKEN : {
66          <ASIG: ":=">                    : DEFAULT
67      |   <DOSPUNTOS: ":">                : DEFAULT
68      |   <MAS: "+">                      : DEFAULT
69      |   <POR: "*">                      : DEFAULT
70      |   <ENTRE: "/">                    : DEFAULT
71      |   <MENOS: "-">                    : DEFAULT
72      |   <LPAREN: "(">                   : DEFAULT
73      |   <RPAREN: ")">                   : DEFAULT
74      |   <LLLAVE: "{">                   : DEFAULT
75      |   <RLLAVE: "}">                   : DEFAULT
76      |   <LCOR: "[">                     : DEFAULT
77      |   <RCOR: "]">                     : DEFAULT
78      |   <MAYOR: ">">                    : DEFAULT
79      |   <MENOR: "<">                    : DEFAULT
80      |   <IGUAL: "=">                    : DEFAULT
81      |   <MAI: ">=">                     : DEFAULT
82      |   <MEI: "<=">                     : DEFAULT
83      |   <DIF: "!=">                     : DEFAULT
84      |   <AND: "AND">                    : DEFAULT
85      |   <OR: "OR">                      : DEFAULT
86      |   <NOT: "NOT">                    : DEFAULT
87      |   <IF: "IF">                      : DEFAULT
88      |   <WHILE: "WHILE">                : DEFAULT
89      |   <REPEAT: "REPEAT">              : DEFAULT
```

```
90      |   <CASO: "CASO">                  : DEFAULT
91      |   <THEN: "THEN">                  : DEFAULT
92      |   <ELSE: "ELSE">                  : DEFAULT
93      |   <FIN: "FIN">                    : DEFAULT
94      |   <DO: "DO">                      : DEFAULT
95      |   <UNTIL: "UNTIL">                : DEFAULT
96      |   <CASE: "CASE">                  : DEFAULT
97      |   <OF: "OF">                      : DEFAULT
98      |   <OTHERWISE: "OTHERWISE">        : DEFAULT
99  }
100 <DEFAULT, INICIO_LINEA> TOKEN [IGNORE_CASE] : {
101         <ID: ["A"-"Z", "_"](["A"-"Z", "0"-"9", "_"])*> : DEFAULT
102 }
103 <DEFAULT, INICIO_LINEA> SKIP : {
104     <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
105                       : DEFAULT
106 }
107 /*
108 gramatica ::= ( sentFinalizada )*
109 */
110 void gramatica():{}{
111     (sentFinalizada())*
112 }
113 /*
114 sentFinalizada ::=      ID ASIG expr ';'
115         |   IF cond THEN sentFinalizada [ ELSE sentFinalizada ] FIN IF ';'
116         |   WHILE cond DO sentFinalizada FIN WHILE ';'
117         |   REPEAT sentFinalizada UNTIL cond ';'
118         |   '{' ( sentFinalizada )* '}'
119         |   CASE expr OF ( CASO expr ':' sentFinalizada )*
120                             [ OTHERWISE sentFinalizada ] FIN CASE ';'
121         |   error ';'
122 */
123 void sentFinalizada():{
124     String s;
125     String e, ei;
126     BloqueCondicion c;
127     String etqFinIf, etqInicioWhile, etqInicioRepeat, etqFinalCaso, etqFinCase;
128 }{
129     try {
130     (
131         s=id() <ASIG> e=expr()    { System.out.println("\t"+s+"="+e); }
132     |   <IF> c=cond() <THEN>      { usarLabel(c.etqVerdad); }
133             sentFinalizada()      {
134                                     usarGoto(etqFinIf=nuevaEtq());
135                                     usarLabel(c.etqFalso);
136                                   }
137             [<ELSE> sentFinalizada()]
138         <FIN> <IF>                { usarLabel(etqFinIf); }
```

```
139     |   <WHILE>                 { usarLabel(etqInicioWhile=nuevaEtq()); }
140             c=cond() <DO>       { usarLabel(c.etqVerdad); }
141             sentFinalizada()
142         <FIN> <WHILE>           {
143                                   usarGoto(etqInicioWhile);
144                                   usarLabel(c.etqFalso);
145                                 }
146     |   <REPEAT>          { usarLabel(etqInicioRepeat=nuevaEtq()); }
147             sentFinalizada()
148         <UNTIL> c=cond() {
149                             usarLabel(c.etqFalso);
150                             usarGoto(etqInicioRepeat);
151                             usarLabel(c.etqVerdad);
152                           }
153     |   <LLLAVE> gramatica() <RLLAVE>
154     |   <CASE> e=expr() <OF>    { etqFinCase = nuevaEtq(); }
155             (<CASO> ei=expr() <DOSPUNTOS>     {
156                                                 System.out.println("\tif "+ e+"!="+ei
157                                                  +"goto "+ (etqFinalCaso=nuevaEtq()));
158                                                         }
159                 sentFinalizada()   {
160                                      usarGoto(etqFinCase);
161                                      usarLabel(etqFinalCaso);
162                                    }
163             )*
164             [<OTHERWISE> sentFinalizada()]
165         <FIN> <CASE>                { usarLabel(etqFinCase); }
166     ) <PUNTOYCOMA>
167     }catch(ParseException x){
168         System.out.println(x.toString());
169         Token t;
170         do {
171             t = getNextToken();
172         } while (t.kind != PUNTOYCOMA);
173     }
174 }
175 /*
176 expr ::= term (('+'|'-') term)*
177 */
178 String expr():{
179     String t1, t2;
180 }{
181     t1=term()     ( <MAS>   t2=term()    { t1=usarOpAritmetico(t1, t2, "+"); }
182                   | <MENOS> t2=term() { t1=usarOpAritmetico(t1, t2, "-"); }
183                   )*    { return t1; }
184 }
185 /*
186 term ::= fact (('*'|'/') fact)*
187 */
```

```
188 String term():{
189       String f1, f2;
190 }{
191       f1=fact()       ( <POR>   f2=fact()     { f1=usarOpAritmetico(f1, f2, "*"); }
192                       | <ENTRE> f2=fact()   { f1=usarOpAritmetico(f1, f2, "/"); }
193                       )*    { return f1; }
194 }
195 /*
196 fact ::= ('-')* ( ID | NUMERO | '(' expr ')' )
197 */
198 String fact():{
199       String s, e, temporal;
200       boolean negado = false;
201 }{    (<MENOS> {negado = !negado;} )*
202       (
203             s=id()          { temporal = s; }
204       |     s=numero()  { temporal = s; }
205       |     <LPAREN> e=expr() <RPAREN> { temporal = e; }
206       ) { if (negado) temporal=usarOpAritmetico("", temporal, "-");
207          return temporal;
208        }
209 }
210 /*
211 cond ::= condTerm (OR condTerm)*
212 */
213 BloqueCondicion cond():{
214       BloqueCondicion c1, c2;
215 }{
216       c1=condTerm()   (     <OR>    { System.out.println("label "+ c1.etqFalso); }
217                             c2=condTerm()    {
218                                   System.out.println("label "+ c1.etqVerdad);
219                                   System.out.println("\tgoto "+ c2.etqVerdad);
220                                   c1 = c2;
221                                         }
222                       )* { return c1; }
223 }
224 /*
225 condTerm ::= condFact (AND condFact)*
226 */
227 BloqueCondicion condTerm():{
228       BloqueCondicion c1, c2;
229 }{
230       c1=condFact()   (     <AND>  { System.out.println("label "+ c1.etqVerdad); }
231                             c2=condFact() {
232                                   System.out.println("label "+ c1.etqFalso);
233                                   System.out.println("\tgoto "+ c2.etqFalso);
234                                   c1 = c2;
235                                   }
236                 )*    { return c1; }
```

```
237 }
238 /*
239 condFact ::= (NOT)* ( condSimple | '[' cond ']' )
240 */
241 BloqueCondicion condFact():{
242       BloqueCondicion c1;
243       boolean negado = false;
244 }{    (<NOT> {negado = !negado;} )*
245       (     c1=condSimple()
246       |     <LCOR> c1=cond() <RCOR>
247       ) { if (negado) intercambiarCondicion(c1);
248           return c1;
249         }
250 }
251 /*
252 condSimple ::= expr ('>'|'<'|'='|'>='|'<='|'!=') expr)*
253 */
254 BloqueCondicion condSimple():{
255       String e1, e2;
256 }{
257       e1=expr()     (
258                     <MAYOR> e2=expr()  { return usarOpRelacional(e1, e2, ">"); }
259               |     <MENOR> e2=expr()  { return usarOpRelacional(e1, e2, "<"); }
260               |     <IGUAL> e2=expr()  { return usarOpRelacional(e1, e2, "="); }
261               |     <MAI> e2=expr()    { return usarOpRelacional(e1, e2, ">="); }
262               |     <MEI> e2=expr()    { return usarOpRelacional(e1, e2, "<="); }
263               |     <DIF> e2=expr()    { return usarOpRelacional(e1, e2, "!="); }
264               )
265 }
266 String id():{}{
267       <ID>      { return token.image; }
268 }
269 String numero():{}{
270       <NUMERO> { return usarOpAritmetico(token.image, "", ""); }
271 }
```

Las funciones de las líneas 21 a 50 permiten clarificar las acciones semánticas repartidas entre las reglas BNF. Por último, las etiquetas declaradas en la línea 127 podrían haberse fusionado en una sola, ya que nunca van a utilizarse dos de ellas simultáneamente; no se ha hecho así para mejorar la legibilidad del código.

Capítulo 9

Gestión de la memoria en tiempo de ejecución

9.1 Organización de la memoria durante la ejecución

Como ya se comentó en los primeros epígrafes, para que un programa se ejecute sobre un sistema operativo, es necesaria la existencia de un cargador que suministra al programa un bloque contiguo de memoria sobre el cual ha de ejecutarse. Por tanto, el código del programa resultante de la compilación debe organizarse de forma que haga uso de este bloque, por lo que el compilador debe incorporar al programa objeto todo el código necesario para ello.

Las técnicas de gestión de la memoria durante la ejecución del programa difieren de unos lenguajes a otros, e incluso de unos compiladores a otros. En este capítulo se estudia la gestión de la memoria que se utiliza en lenguajes imperativos como Fortran, Pascal, C, Modula-2, etc. La gestión de la memoria en otro tipo de lenguajes (funcionales, lógicos, etc.) es, en general, diferente de la organización que aquí se plantea.

Para lenguajes imperativos, los compiladores generan programas que tendrán en tiempo de ejecución una organización de la memoria similar (a grandes rasgos) a la que aparece en la figura 9.1.

En este esquema se distinguen claramente las secciones de:

- el Código
- la Zona de Datos de Tamaño Fijo
- la Pila o *stack*
- el Montón o *heap*

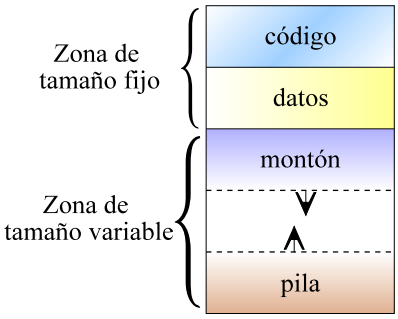


Figura 9.1. Estructura de la memoria durante la ejecución de un programa compilado

9.2 Zona de código

Es la zona donde se almacenan las instrucciones del programa ejecutable en código máquina, y también el código correspondiente a los procedimientos y funciones que utiliza. Su tamaño y su contenido se establecen en tiempo de compilación y por ello se dice que su tamaño es fijo en tiempo de ejecución.

De esta forma, el compilador, a medida que va generando código, lo va situando secuencialmente en esta zona, delimitando convenientemente el inicio de cada función, procedimiento y programa principal (si lo hubiera, como sucede en lenguajes como Pascal o Modula-2).

El programa ejecutable final estará constituido por esta zona junto con información relativa a las necesidades de memoria para datos en tiempo de ejecución, tanto del bloque de tamaño fijo como una estimación del tamaño del bloque dinámico de datos. Esta información será utilizada por el cargador (*linker*) en el momento de hacer la carga del ejecutable en memoria principal para proceder a su ejecución.

9.2.1 *Overlays*

Algunos compiladores fragmentan el código del programa objeto usando *overlays* o "solapas", cuando la memoria principal disponible es inferior al tamaño del programa completo. Estos *overlays* son secciones de código objeto que se almacenan en ficheros independientes y que se cargan en la memoria central dinámicamente, es decir, durante la ejecución del programa. Se les llama "solapas" porque dos *overlays* pueden ocupar el mismo trozo de memoria en momentos de tiempo diferentes, solapándose. Para hacer más eficiente el uso de la memoria, los *overlays* de un programa se agrupan en zonas y módulos, cada uno de los cuales contiene un conjunto de funciones o procedimientos completos. La figura 9.2 muestra cómo se ubican varios

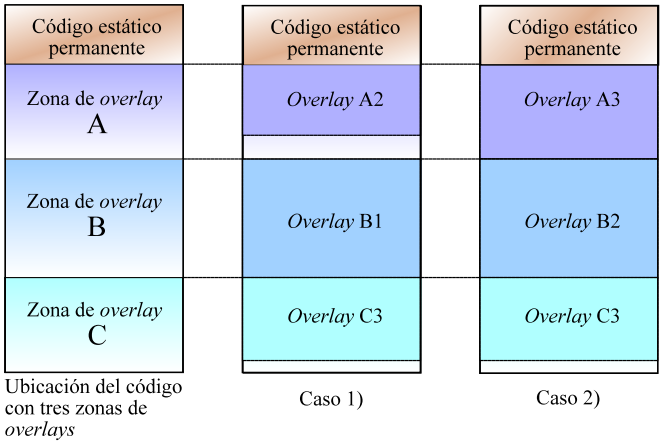


Figura 9.2. Ejemplos de configuraciones de la zona de código utilizando *overlays*

overlays en distintas zonas de memoria estática.

Durante el tiempo de ejecución sólo uno de los *overlays* de cada una de las zonas de *overlay* puede estar almacenado en memoria principal. El compilador reserva en la sección de código una zona contigua de memoria para cada conjunto de *overlays*. El tamaño de esta zona debe ser igual al del mayor módulo que se cargue sobre ella. Es función del programador determinar cuantas zonas de *overlay* se definen, qué funciones y procedimientos se encapsulan en cada módulo de *overlay*, y cómo se organizan estos módulos para ocupar cada una de las zonas de *overlay*. Una restricción a tener en cuenta es que las funciones de un módulo no deben hacer referencia a funciones de otro módulo del mismo *overlay*, ya que nunca estarán simultáneamente en memoria.

Evidentemente, el tiempo de ejecución de un programa estructurado con *overlays* es mayor que si no tuviese *overlays* y todo el código estuviese residente en memoria, puesto que durante la ejecución del programa es necesario cargar cada módulo cuando se realiza una llamada a alguna de las funciones que incluye. También es tarea del programador diseñar la estructura de *overlays* de manera que se minimice el número de estas operaciones. La técnica de *overlays* no sólo se utiliza cuando el programa a compilar es muy grande en relación con la disponibilidad de memoria del sistema, sino también cuando se desea obtener programas de menor tamaño que deben coexistir con otros del sistema operativo cuando la memoria es escasa.

9.3 Zona de datos

Los datos que maneja un programa se dividen actualmente en tres grandes bloques:

- Uno dedicado a almacenar las variables globales accesibles por cualquier línea de código del programa. Este bloque es la Zona de Datos de Tamaño Fijo.
- Otro dedicado a almacenar las variables locales a cada función y procedimiento. Éstas no pueden almacenarse en el bloque anterior por dos motivos principales: a) no es necesario almacenar las variables de una función hasta el momento en que ésta es invocada y, una vez que finaliza su ejecución, tampoco; y b) durante la ejecución de una función recursiva deben almacenarse varias instancias de sus variables locales, concretamente tantas como invocaciones a esa misma función se hayan producido. Este bloque es la Pila.
- Un último bloque dedicado a almacenar los bloques de memoria gestionados directamente por el usuario mediante sentencias **malloc/free**, **new/dispose**, etc. La cantidad de memoria requerida para estos menesteres varía de ejecución en ejecución del programa, y los datos ubicados no son locales a ninguna función, sino que perduran hasta que son liberados. Este bloque es el Montón.

9.3.1 Zona de Datos de Tamaño Fijo

La forma más fácil de almacenar el contenido de una variable en memoria en tiempo de ejecución es en memoria estática o permanente a lo largo de toda la ejecución del programa. No todos los objetos (variables) pueden ser almacenados estáticamente. Para que un objeto pueda ser almacenado en memoria estática, su tamaño (número de *bytes* necesarios para su almacenamiento) ha de ser conocido en tiempo de compilación. Como consecuencia de esta condición no podrán almacenarse en memoria estática:

- Las variables locales correspondientes a procedimientos o funciones recursivas, ya que en tiempo de compilación no se sabe el número de veces que estas variables serán necesarias.
- Las estructuras dinámicas de datos tales como listas, árboles, etc. ya que el número de elementos que la forman no es conocido hasta que el programa se ejecuta.

Por tanto, las variables locales a una función son datos de tamaño definido, pero para los que no se sabe el número de ocurrencias del ámbito del que dependen, por lo que deben almacenarse siguiendo algún criterio dinámico. Tampoco es posible almacenar en esta zona a las estructuras eminentemente dinámicas, esto es, las que están gestionadas a través de punteros mediante creación dinámica de espacio libre: listas dinámicas, grafos dinámicos, etc. Estos datos son de tamaño indefinido por lo que no es posible guardarlos en la zona de datos de tamaño fijo. Aunque pueda pensarse que el tamaño del código de una función recursiva también depende del número de veces que la función se llama a sí misma, esto no es cierto: las instrucciones a ejecutar son siempre las mismas, y lo que cambia son los datos con los que se trabaja, o sea, las variables locales.

Las técnicas de asignación de memoria estática son sencillas, tal y como ilustra la figura 9.3. A partir de una posición señalada por un puntero de referencia

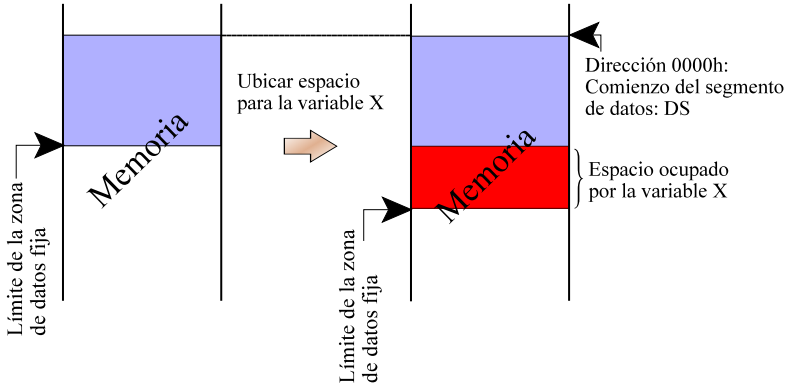


Figura 9.3. Asignación de un bloque de memoria para una variable global **X**

(límite de la zona de datos fija) se aloja la variable global **X**, y se avanza el puntero tantos *bytes* como sean necesarios para almacenar dicha variable. En otras palabras, a cada variable se le asigna una dirección y un trozo de memoria a partir de ella acorde con el tipo de la variable; estos trozos se van alojando secuencialmente a partir del comienzo del área de datos de tamaño fijo. Por ejemplo, si encontramos la declaración de Modula-2:

```
a, b : INTEGER;
c : ARRAY [1..20] OF REAL;
c : CHAR;
d : REAL;
```

y suponiendo que un INTEGER ocupa 2 bytes, un REAL ocupa 5 y un CHAR ocupa 1, y suponiendo que el comienzo del área de datos se encuentra en la posición 0000h (esta dirección es relativa al segmento de datos DS que, realmente, puede apuntar a cualquier dirección de memoria y que será inicializado convenientemente por el cargador), entonces se tiene la siguiente asignación de direcciones y tamaños:

Variable	Tamaño	Dir. comienzo
a	2 bytes	0000h
b	2 bytes	0002h
c	100 bytes	0004h
d	1 byte	0068h
e	5 byte	0069h

Esta asignación de memoria se hace en tiempo de compilación y los objetos están vigentes y son accesibles desde que comienza la ejecución del programa hasta que termina. La dirección asociada a cada variable global es constante y relativa al segmento de datos, o sea, siempre la misma a partir del punto de inicio de memoria en que se ha cargado el programa (punto que puede variar de una ejecución a otra).

En los lenguajes que permiten la existencia de subprogramas, y siempre que todos los objetos de estos subprogramas puedan almacenarse estáticamente (por ejemplo en Fortran-IV que no permite la recursión) se aloja en la memoria estática un registro de activación correspondiente a cada uno de los subprogramas, con una estructura como la de la figura 9.4.a). Estos registros de activación contendrán las variables locales, parámetros formales y valor devuelto por la función, y se distribuyen linealmente en la zona de datos de tamaño fijo tal como indica la figura 9.4.b).

Dentro de cada registro de activación las variables locales se organizan en secuencia. Existe un solo registro de activación para cada procedimiento y por tanto no están permitidas las llamadas recursivas. El proceso que se sigue cuando un procedimiento **P** llama a otro **Q** es el siguiente:

- **P** (el llamador) evalúa los parámetros reales de invocación, en caso de que se trate de expresiones complejas, usando para ello una zona de memoria temporal para el almacenamiento intermedio. Por ejemplo, si la llamada a **Q** es "Q((3*5)+(2*2),7)" las operaciones previas a la llamada propiamente dicha en código máquina han de realizarse sobre alguna zona de memoria temporal.

(En algún momento debe haber una zona de memoria que contenga el valor intermedio 15, y el valor intermedio 4 para sumarlos a continuación). En caso de utilización de memoria estática, esta zona de temporales puede ser común a todo el programa, ya que su tamaño puede deducirse en tiempo de compilación.

- **Q** inicializa sus variables y comienza su ejecución.

Dado que las variables están permanentemente en memoria es fácil implementar la propiedad de que conserven o no su contenido para cada nueva llamada (comportamiento parecido al del modificador **static** del lenguaje C).

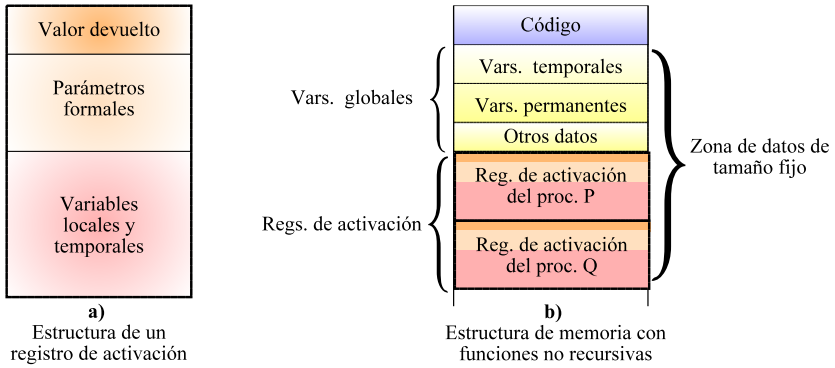


Figura 9.4. Estructura de un registro de activación asociado a una función no recursiva. Cada función o procedimiento tiene asociado un registro de activación de tamaño diferente, en función de cuantas variables locales posea, del tipo del valor devuelto, etc.

9.3.2 Pila (*Stack*)

Un lenguaje con estructura de bloques es aquél que está compuesto por módulos o trozos de código cuya ejecución es secuencial; estos módulos a su vez pueden contener en su secuencia de instrucciónes llamadas a otros módulos, que a su vez pueden llamar a otros submódulos y así sucesivamente.

La aparición de este tipo de lenguajes trajo consigo la necesidad de técnicas de alojamiento en memoria más flexibles, que pudieran adaptarse a las demandas de memoria durante la ejecución del programa debido a las invocaciones recursivas. En estos lenguajes, cada vez que comienza la ejecución de un procedimiento se crea un registro de activación para contener los objetos necesarios para su ejecución, eliminandolo una vez terminada ésta.

Durante una invocación recursiva sólo se encuentra activo (en ejecución) el último procedimiento invocado, mientras que la secuencia de invocadores se encuentra "dormida". Esto hace que, a medida que las funciones invocadas van finalizando, deban reactivarse los procedimientos llamadores en orden inverso a la secuencia de llamadas.

Por esto, los distintos registros de activación asociados a cada bloque deberán colocarse en una pila en la que entrarán cuando comience la ejecución del bloque y saldrán al terminar el mismo. La estructura de los registros de activación varía de unos lenguajes a otros, e incluso de unos compiladores a otros, siendo éste uno de los problemas por los que a veces resulta difícil enlazar los códigos generados por dos compiladores diferentes. En general, los registros de activación de los procedimientos suelen tener algunos de los campos que pueden verse en la figura 9.5.

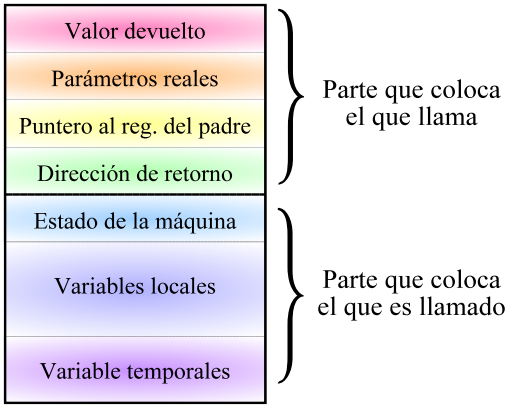


Figura 9.5. Estructura completa de un registro de activación en lenguajes que admiten recursión

En la zona correspondiente al **estado de la máquina** se almacena el contenido que hubiera en los registros del microprocesador antes de comenzar a ejecutarse el procedimiento. Estos valores deberán ser repuestos al finalizar su ejecución. El código encargado de realizar la copia del estado de la máquina es común para todos los procedimientos.

El puntero al registro de activación del padre permite el acceso a las variables declaradas en otros procedimientos dentro de los cuales se encuentra inmerso aquél al que pertenece este registro de activación. Por ejemplo, supongamos un bloque de programa como el siguiente:

```
PROCEDURE A1
VAR
    A1_a, A1_b : INTEGER;
    PROCEDURE B1
    VAR
        B1_c, B1_d : INTEGER;
    BEGIN
        ...
    END B1;
    PROCEDURE B2
    VAR
        B2_e, B2_f : INTEGER;
```

```
            PROCEDURE C2
            VAR
                C2_g, C2_h : INTEGER;
            BEGIN
                ...
                CALL B1;
                ...
            END C2;
        BEGIN
            ...
            CALL C2;
            ...
        END B2;
    BEGIN (* Programa principal *)
        ...
        CALL B2;
        ...
    END;
```

El puntero al registro de activación del padre se utiliza porque un procedimiento puede hacer uso de las variables locales de los procedimientos en los que se halla declarado. En este ejemplo, C2 puede hacer uso de las variables:

- C2_g y C2_h (que son suyas).
- B2_e y B2_f (que son de su padre, aquél en el que C2 está declarado).
- A1_a y A1_b (que son del padre de su padre).

pero no puede hacer uso ni de B1_c ni de B1_d, ya que C2 no está inmerso en B1.

Al igual que en la zona de datos de tamaño fijo, los registros de activación contienen espacio para almacenar los parámetros reales (valores asociados a las variables que aparecen en la cabecera) y las variables locales, (las que se definen dentro del bloque o procedimiento) así como una zona para almacenar el valor devuelto por la función y una zona de valores temporales para el cálculo intermedio de expresiones.

Cualesquiera dos bloques o procedimientos diferentes, suelen tener registros de activación de tamaños diferentes. Este tamaño, por lo general, es conocido en tiempo de compilación ya que se dispone de información suficiente sobre el espacio ocupado por los objetos que lo componen. En ciertos casos esto no es así como por ejemplo ocurre en C cuando se utilizan *arrays* de longitud indeterminada. En estos casos el registro de activación debe incluir una zona de desbordamiento al final cuyo tamaño no se fija en tiempo de compilación sino sólo cuando realmente llega a ejecutarse el procedimiento. Esto complica un poco la gestión de la memoria, por lo que algunos compiladores de bajo coste suprimen esta facilidad.

A pesar de que es imprescindible guardar sucesivas versiones de los datos según se van realizando llamadas recursivas, las instrucciones que se aplican sobre ellos son siempre las mismas, por lo que estas instrucciones sólo es necesario guardarlas una vez. No obstante, es evidente que el mismo código trabajará con diferentes datos en cada invocación recursiva, por lo que el compilador debe generar

código preparado para tal eventualidad, no haciendo referencia absoluta a las variables (excepto a las definidas globalmente cuya ubicación en memoria no cambia en el transcurso de la ejecución del programa), sino referencia relativa en función del comienzo de su registro de activación.

El procedimiento de gestión de la pila cuando un procedimiento **P** llama a otro procedimiento **Q**, se desarrolla en dos fases; la primera de ellas corresponde al código que se incluye en el procedimiento **P** antes de transferir el control a **Q**, y la segunda, al código que debe incluirse al principio de **Q** para que se ejecute cuando reciba el control. Un ejemplo de esta invocación puede venir dado por un código como:

```
PROCEDURE P( /* param. formales de P */ ) {
    VAR
        // var. locales de P
        PROCEDURE Q( /* param. formales de Q */ ) {
            VAR
                // var. locales de Q
            BEGIN
                ...
                Q( ... ); // Invocación recursiva de Q a Q
                ...
        END Q;
    BEGIN
        ...
        x = Q( /* param. reales a Q */ ); // Invocación de P a Q
END P;
```

La primera de estas fases sigue los siguientes pasos:

1.1 El procedimiento que realiza la llamada (**P**) evalúa las expresiones de la invocación, utilizando para ello su zona de variables temporales, y copia el resultado en la zona correspondiente a los parámetros reales del procedimiento que recibe la llamada. Previo a ello deja espacio para que **Q** deposite el valor devuelto.

1.2 El llamador **P** coloca en la pila un puntero al registro de activación del procedimiento en el cual se halla declarado **Q**, con objeto de que éste pueda acceder a las variables no locales declaradas en su padre. Por último, **P** transfiere el control a **Q**, lo que hace colocar la dirección de retorno encima de la pila.

1.3 El receptor de la llamada (**Q**) salva el estado de la máquina antes de comenzar su ejecución usando para ello la zona correspondiente de su registro de activación.

1.4 **Q** inicializa sus variables y comienza su ejecución.

Al terminar **Q** su ejecución se desaloja su registro de activación procediendo también en dos fases. La primera se implementa mediante instrucciones al final del procedimiento que acaba de terminar su ejecución (**Q**), y la segunda en el procedimiento que hizo la llamada (**P**), tras recobrar el control:

2.1 El procedimiento saliente (**Q**) antes de finalizar su ejecución coloca el valor

de retorno al principio de su registro de activación.

2.2 Usando la información contenida en su registro, **Q** restaura el estado de la máquina y coloca el puntero de final de pila en la posición en la que estaba originalmente.

2.3 El procedimiento llamador **P** copia el valor devuelto por el procedimiento invocado **Q** dentro de su propio registro de activación (el de **Q**).

La figura 9.6 ilustra el estado de la pila como consecuencia de realizar la invocación de **P** a **Q**, y luego una sola invocación recursiva de **Q** a **Q**. Nótese en esta figura que la pila crece hacia abajo, como suele dibujarse siempre, ya que crece de las direcciones de memoria superiores a las inferiores.

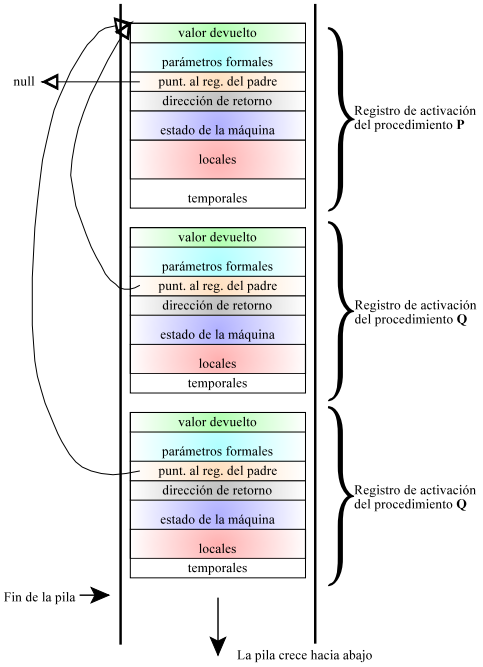


Figura 9.6. Pila de registros de activación cuando una función **P** invoca a otra **Q** y ésta sellama a sí misma una sola vez. Nótese que el padre de **Q** es **P**, ya que **Q** se halla declarada localmente a **P**. **P** no tiene padre puesto que se supone declarada globalmente

Dentro de un procedimiento o función, las variables locales se referencian siempre como direcciones relativas al comienzo de su registro de activación, o bien al comienzo de la zona de variables locales. Por tanto, cuando sea necesario acceder desde un procedimiento a variables definidas de otros procedimientos cuyo ámbito sea accesible, será necesario proveer dinámicamente la dirección de comienzo de las variables de ese procedimiento. Para ello se utiliza dentro del registro de activación el puntero al registro de activación del padre. Este puntero, señalará al comienzo de las variables locales del procedimiento inmediatamente superior (en el que el llamado se encuentra declarado). El puntero de enlace ocupa una posición fija con respecto al

comienzo de sus variables locales. Cuando los procedimientos se llaman a sí mismos recursivamente, el ámbito de las variables impide por lo general que una activación modifique las variables locales de otra activación del mismo procedimiento, por lo que en estos casos el procedimiento inmediato superior será, a efectos de enlace, el que originó la primera activación, tal como puede apreciarse en la figura 9.6. Es importante notar que el concepto de "procedimiento padre" es independiente del de "procedimiento llamador", aunque puede coincidir con éste.

9.3.3 Montón (*heap*)

Cuando el tamaño de un objeto a colocar en memoria puede variar en tiempo de ejecución, no es posible su ubicación en la pila, y mucho menos en la zona de datos de tamaño fijo. Son ejemplos de este tipo de objetos las listas y los árboles dinámicos, las cadenas de caracteres de longitud variable, etc. Para manejar este tipo de objetos el compilador debe disponer de un área de memoria de tamaño variable y que no se vea afectada por la activación o desactivación de procedimientos. Este trozo de memoria se llama montón (del inglés *heap*). En aquellos lenguajes de alto nivel que requieran el uso del *heap*, el compilador debe incorporar en el programa objeto generado, todo el código correspondiente a la gestión de éste; este código es siempre el mismo para todos los programas construidos. Las operaciones básicas que se realizan sobre el *heap* son:

- Alojar: se solicita un bloque contiguo de memoria para poder almacenar un ítem de un cierto tamaño.
- Desalojar: se indica que ya no es necesario conservar la memoria previamente alojada para un objeto y que, por lo tanto, ésta debe quedar libre para ser reutilizada en caso necesario por otras operaciones de alojamiento..

Según sea el programador o el propio sistema el que las invoque, estas operaciones pueden ser explícitas o implícitas respectivamente. En caso de alojamiento explícito, el programador debe incluir en el código fuente, una por una, las instrucciones que demandan una cierta cantidad de memoria para la ubicación de cada dato o registro (por ejemplo, Pascal proporciona la instrucción **new**, C proporciona **malloc**, etc.). La cantidad de memoria requerida en cada operación atómica de alojamiento, puede ser calculada por el compilador en función del tipo correspondiente al objeto que se desea alojar, o bien puede especificarla el programador directamente. El resultado de la función de alojamiento es por lo general un puntero a un trozo contiguo de memoria dentro del *heap* que puede usarse para almacenar el valor del objeto. Los lenguajes de programación imperativos suelen utilizar alojamiento y desalojamiento explícitos. Por el contrario los lenguajes declarativos (lógicos y funcionales) hacen la mayoría de estas operaciones implícitamente debido a los mecanismos que subyacen en su funcionamiento.

La gestión del *heap* requiere técnicas adecuadas que optimicen el espacio que se ocupa (con objeto de impedir la fragmentación de la memoria) o el tiempo de acceso, encontrándose contrapuestos ambos factores como suele suceder casi siempre en

informática. A este respecto, las técnicas suelen dividirse en dos grandes grupos: aquéllas en las que se aloja exactamente la cantidad de memoria solicitada (favorecen la fragmentación), y aquéllas en las que el bloque alojado es de un tamaño parecido, pero generalmente superior, al solicitado, lo que permite que el *heap* sólo almacene bloques de determinados tamaños prefijados en el momento de construir el compilador (favorece la desfragmentación).

Para finalizar, una de las características más interesantes de lenguajes imperativos muy actuales, como Java, es que incluyen un recolector automático de basura (*garbage collection*). El recolector de basura forma parte del gestor del *heap*, y lleva el control sobre los punteros y los trozos de memoria del *heap* que son inaccesibles a través de las variables del programa, ya sea directa o indirectamente, esto es, de los trozos de memoria alojados pero que no están siendo apuntados por nadie y que, por lo tanto, constituyen basura. Mediante este control pueden efectuar de manera automática las operaciones de desalojamiento, liberando al programador de realizar explícitamente esta acción. Entre las técnicas utilizadas para implementar los recolectores de basura podemos citar los contadores de referencia, marcar y barrer, técnicas generacionales, etc.

ISBN 84-689-1037-6
9 788468 910376

UNIVERSIDAD
DE MÁLAGA

El presente volumen introduce al lector en uno de los aspectos más potentes de la informática tradicional: el análisis y comprensión de ficheros de texto. Las técnicas y herramientas que aquí se examinan, se encuentran ampliamente difundidas y no están orientadas exclusivamente a la construcción de compiladores e intérpretes, sino que establecen un marco general con el que el informático puede analizar textos con cualquier otro objetivo. Cualquier transformación semántica imaginable computacionalmente puede hacerse realidad, desde el procesamiento de datos tabulares hasta la conversión de subtítulos en películas para ordenador, pasando por la transformación de programas fuente, generación de índices analíticos, de materias, etc.

Los primeros capítulos presentan una panorámica general de los conceptos básicos que sustentan estas técnicas, a la vez que se exponen las herramientas Lex y Yacc y sus contrapartidas JFLex y Cup que generan analizadores sintácticos y lexicográficos en lenguaje Java. También se estudia con profusión el funcionamiento de La herramienta JavaCC como representante más extendida de los generadores de análisis sintácticos descendentes.

Los capítulos siguientes se centran en la utilización de estos metaprogramas introduciendo técnicas generales de gestión semántica (tablas de símbolos, asociación de atributos, mejora de gramáticas, etc.) aplicadas a las diferentes fases que se siguen en la construcción de un traductor. El texto culmina con una introducción al manejo de la recursividad y de la memoria dinámica en tiempo de ejecución.